UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS RUSSAS

CURSO DE GRADUAÇÃO EM ENGENHARIA DE SOFTWARE

MATHEUS DE SOUZA OLIVEIRA

INSPEÇÃO E TESTES EM UM SISTEMA DE ROTEIRIZAÇÃO DE VEÍCULOS: UM ESTUDO DE CASO

RUSSAS

2018

MATHEUS DE SOUZA OLIVEIRA

INSPEÇÃO E TESTES EM UM SISTEMA DE ROTEIRIZAÇÃO DE VEÍCULOS: UM ESTUDO DE CASO

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Software do Campus Russas da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Software.

Orientador: Prof. Dr. Dmontier Pinheiro Aragão Jr.

RUSSAS

2018

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

O48i Oliveira, Matheus de Souza.
Inspeção e testes em um sistema de roteirização de veículos : Um estudo de caso / Matheus de Souza Oliveira. – 2018.
53 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Russas, Curso de Engenharia de Software, Russas, 2018.
Orientação: Prof. Dr. Dmontier Pinheiro Aragão Junior.

1. Inspeção. 2. Teste. 3. V&V. 4. Estudo de caso. 5. Roteirização de Veículos. I. Título.
CDD 005.1

MATHEUS DE SOUZA OLIVEIRA

INSPEÇÃO E TESTES EM UM SISTEMA DE ROTEIRIZAÇÃO DE VEÍCULOS: UM ESTUDO DE CASO

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Software do Campus Russas da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Software.

Aprovada em:

BANCA EXAMINADORA

Prof. Dr. Dmontier Pinheiro Aragão Jr. (Orientador)
Universidade Federal do Ceará (UFC)

Profa. Dra. Valéria Lelli Leitão Dantas
Universidade Federal do Ceará (UFC)

Profa. Dra. Anna Beatriz dos Santos Marques
Universidade Federal do Ceará (UFC)

Dedico esta monografia a minha mãe, namorada e amigos por todo apoio durante a minha trajetória e em especial a minha avó Ermelina Leoncio Pereira por ser um exemplo de dedicação, respeito, amizade, amor, enfim, de vida a todos nós.

AGRADECIMENTOS

Em primeiro lugar, gostaria de agradecer a Deus, por ter me dado o presente da vida, iluminando meu caminho para que pudesse lutar e concluir mais uma etapa do interminável caminho da evolução como ser humano.

Aos meus pais e toda a minha família pelo apoio e carinho que me foram dados em todas as fases de minha vida.

A minha namorada Karoline, que sempre esteve ao meu lado me dando força, carinho, amor e apoio. Te Amo!

Aos meus colegas que tive o prazer de conhecer e compartilhar grandes momentos de alegria e aprendizado. Em especial aos que se tornaram verdadeiros amigos nessa caminhada.

Finalmente, agradeço a todos os professores do curso de graduação, em especial ao meu orientador Dmontier Pinheiro Aragão Jr. por todas as orientações, conselhos e incentivos durante as conversas, reuniões e aulas.

“O futuro vai mostrar os resultados e julgar cada um segundo as suas realizações.”

(Nikola Tesla)

RESUMO

Durante o desenvolvimento de um *software* grandes esforços são aplicados pelos programadores para obter um código-fonte de qualidade. Porém, um *software* de qualidade depende de diversos fatores relacionados a sua implementação, tais como valores e custos de desenvolvimento, boas práticas de codificação, processos organizacionais e esforço com atividades de identificação de defeitos. A indústria da Tecnologia da Informação está cada vez mais exigente quanto à qualidade do *software* utilizado em seus negócios, e para atingir esse nível de excelência, o desenvolvimento das aplicações devem seguir técnicas como as de *Verificação e Validação* (V&V), que possibilitam a identificação de falhas do *software*, principalmente quando já está em produção. A aquisição e manutenção desses sistemas geram custos, e um erro pode causar perdas significativas para os negócios do cliente como também para o provedor do serviço. Considerando a importância do sistema para os negócios, o processo de manutenção deve ser contínuo e organizado, de forma a garantir que as correções não introduzam novos erros e nem degradem o código-fonte, diminuindo a manutenibilidade do *software*. Para isso, dentre as técnicas de V&V que podem ser utilizadas para mitigar tais problemas, destacam-se as inspeções e testes, que são verificações estáticas e dinâmicas de *software*. Dessa forma, este trabalho teve por objetivo avaliar o impacto das inspeções automatizadas de código-fonte através de ferramentas de verificação estática e também de técnicas dinâmicas como testes unitários e testes funcionais. Para isso, adotou-se a metodologia de estudo de caso, onde essas técnicas foram aplicadas a um *Sistema de Roteirização de Veículos* (SRV). Como resultados obtidos, foi possível desenvolver e utilizar uma ferramenta de teste de caixa-preta para validação do SRV, aplicar e controlar através de métricas as refatorações indicadas pelas ferramentas de análise estática e também desenvolver, gerar e executar automaticamente testes unitários com apoio de ferramentas automatizadas. Além disso, foi possível desenvolver um modelo de processo para aplicação das técnicas, de forma que essas possam ser utilizadas em conjunto permitindo um melhor aproveitamento dos seus benefícios. Por fim, como conclusão do trabalho, chegou-se ao resultado de que as técnicas juntas alcançaram uma melhoria significativa na qualidade do sistema, principalmente no que se diz respeito a manutenibilidade do *software*.

Palavras-chave: Inspeção. Teste. V&V. Estudo de Caso. Roteirização de Veículos.

ABSTRACT

During the development of a software the most part of efforts are applied by the programmers to obtain a quality source code. However, quality software depends on several factors related to its implementation, such as development values and costs, good coding practices, organizational processes and effort with defect identification activities. The Information Technology industry is increasingly demanding about the quality of software used in its business, and to achieve this level of excellence, application development must follow techniques such as Verification and Validation (V&V), which enable the identification of software failures, mainly when it is already in production. Acquiring and maintaining these systems generates costs, and an error can cause significant losses to the customer's business as well as to the service provider. Considering the importance of the system for business, the maintenance process must be continuous and organized, in order to ensure that the corrections do not introduce new errors nor degrade the source code, reducing the maintainability of the software. For this, among the V&V techniques that can be used to mitigate such problems, stand out the inspections and tests, which are static and dynamic verifications of software. Thus, the objective of this work was to evaluate the impact of automated source code inspections through static verification tools as well as dynamic techniques such as unit tests and functional tests. For this, a methodology of case study was adopted, where these techniques were applied to a Vehicle Routing System. As a result, it was possible to develop and use a black-box test tool to validate Vehicle Routing System, to apply and to control by means of metrics the refactorings indicated by the static analysis tools, and also to develop, generate and execute automatically unit tests with tool automated support. In addition, it was possible to develop a process model for applying the techniques, so that they can be used together allowing a better use of its benefits. Finally, as a conclusion of the work, reached the result that the techniques together achieved a significant improvement in the quality of the system, especially with regard to the maintainability of the software.

Keywords: Inspection. Test. V&V. Case study. Vehicle Routing.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

DDRE	*Dynamic Defect Removal Efficiency ou Eficiência Dinâmica de Remoção de Defeitos*
DKLOC	*Defeitos por KLOC*
DM	*Defeitos por Método*
DRDE	*Defect Reduction Efficiency ou Eficiência de Redução de Defeitos*
DRE	*Defect Removal Efficiency ou Eficiência de Remoção de Defeitos*
ECT	*Erros por caso de teste*
QDP	*Quantidade de Defeitos Encontradas no Produto*
SQA	*Software Quality Assurance*
SRV	*Sistema de Roteirização de Veículos*
TDE	*Tipos de Defeitos Encontrados*
TI	*Tecnologia da Informação*
V&V	*Verificação e Validação*

SUMÁRIO

1 INTRODUÇÃO

Considerando a impossibilidade de garantir a qualidade de um produto de *software* sem técnicas que ajudem a prevenir, encontrar e corrigir defeitos, não se pode inserir qualidade em um produto mal construído apenas com testes, mas também não é possível construir um produto de qualidade sem eles (PEZZÈ; YOUNG, 2008).

Para que erros no *software* não perdurem, isto é, que sejam descobertos antes do *software* entrar em produção, existem uma série de atividades que, em conjunto são conhecidas como "Verificação e Validação" ou "V&V", possuem a finalidade de garantir que tanto o modo pelo qual o *software* está sendo construído quanto o produto em si estejam em conformidade com o especificado (TERRA; BIGONHA, 2008). Contudo, as técnicas para mitigar a quantidade de defeitos de um sistema vão muito além de testar, e para isso é importante destacar conceitos como o de verificação estática e dinâmica que fazem parte do processo de *Verificação e Validação* (V&V).

A verificação estática preocupa-se em manter os erros fora do *software*, também conhecida como inspeção, pode ser aplicada a qualquer artefato do sistema, inclusive no código-fonte. É possível então realizar análises destes artefatos do *software* de maneira que se identifique prováveis anomalias no projeto, sem mesmo ter a necessidade de executar o sistema ou componente para isso. Segundo Delamaro *et al.* (2017), técnicas estáticas são aquelas que não requerem a execução ou mesmo a existência de um programa ou modelo executável para serem conduzidas. Já a verificação dinâmica busca encontrar defeitos através da ação de executar o *software* sobre determinadas condições específicas, a fim de encontrar situações em que ele tenha um comportamento diferente do esperado, essa caracteriza-se por testes. Segundo Terra e Bigonha (2008), testes de *software* envolvem executar uma implementação do sistema com dados de teste, são examinadas as saídas e seu comportamento operacional para verificar se seu desempenho está conforme necessário. De forma análoga, para Delamaro *et al.* (2017), técnicas dinâmicas são aquelas que se baseiam na execução de um programa ou de um modelo.

Ainda nesta mesma linha de considerações, vale ressaltar que vários estudos e experimentos mostram que as inspeções são mais eficientes na descoberta de defeitos do que os testes. Segundo Sommerville (2010), inspeções conseguem considerar atributos de qualidade de um programa que nem mesmos são requisitos funcionais, como a conformidade com padrões, portabilidade e manutenibilidade, encontrando ineficiências, algoritmos inadequados e um estilo de programação pobre que poderiam tornar o código de difícil manutenção e atualização. Porém

o teste ainda é a técnica mais utilizada e deve ser complementadas pelas inspeções, pois ambas tem suas vantagens e desvantagens, mas quando utilizadas em conjunto fornecem uma cobertura maior na verificação de erros do sistema.

1.1 Justificativa

A indústria da *Tecnologia da Informação* (TI) está cada vez mais exigente quanto à qualidade do *software* utilizado em seus negócios. Para atingir esse nível de excelência, o desenvolvimento das aplicações deve seguir técnicas que possibilitem a identificação de falhas do *software*, principalmente quando o sistema já está em produção.

Atualmente, uma grande parte dos sistemas que entram em produção apresentam erros (TERRA; BIGONHA, 2008). Para diminuir esse problema, as atividades de V&V se tornam uma peça crucial para o desenvolvimento de produtos de alta qualidade (TERRA; BIGONHA, 2008).

As organizações de desenvolvimento de *software* enfrentam o difícil problema de produzir *software* de alta qualidade e poucos defeitos no prazo e com custo dentro do orçamento (WILKERSON *et al.*, 2012). Por exemplo, em um estudo do governo dos EUA Tassey (2002) estimou que os defeitos de *software* estavam custando à economia dos EUA cerca de US$59,5 bilhões por ano.

Um dos ramos da indústria que depende muito da TI é a logística, que tem como uma das suas prioridades o transporte adequado e rápido das suas mercadorias. Em um mundo globalizado, a qualidade e os prazos de entrega se tornam ainda mais importantes para se obter vantagens em relação aos concorrentes. Para atingir esse objetivo, as empresas usam como ferramenta o *Sistema de Roteirização de Veículos* (SRV), diminuindo os custos, aumentando os lucros e satisfação dos clientes.

De acordo com Silva Melo e Ferreira Filho (2001):

> Adquirir um sistema de roteirização pode permitir ganhos significativos, tanto do ponto de vista financeiro, com a redução dos custos operacionais, quanto em termos da qualidade do serviço permitindo maior quantidade e fidelidade de clientes, ganhos estes de grande importância para a melhor integração da cadeia de suprimentos e, consequentemente, para a obtenção de vantagens competitivas.

No entanto, a aquisição e manutenção desses sistemas geram custos, e um erro pode causar perdas significativas para os negócios do cliente como também para o provedor do serviço. Considerando a importância do sistema para os negócios, o processo de manutenção deve ser

contínuo e organizado, de forma a garantir que as correções não introduzam novos erros e nem degradem o código-fonte. Isso diminui a manutenibilidade do *software*, essa que por sua vez é um dos principais atributos de qualidade, pois um sistema difícil de manter logo irá se tornar obsoleto.

Em virtude da diversidade de critérios de teste existentes, saber quais deles devem ser utilizados ou como utilizá-los de forma complementar a fim de obter o melhor resultado com o menor custo é uma questão complicada. Assim, pesquisadores procuram definir técnicas, critérios e ferramentas que possibilitem a aplicação de tais atividades de maneira sistemática, com alta qualidade e custo reduzido.

A escolha de um estudo de caso é de suma importância para qualquer análise empírica em engenharia de *software* (ROJAS *et al.*, 2016). Segundo Wiklund *et al.* (2017), após 2004, houve um aumento de pesquisas nas áreas de engenharia e testes de *software*, que provavelmente pode ser explicado por um interesse crescente em estudos empíricos nessas áreas. Portanto, julga-se necessário difundir princípios e conceitos relacionados à aplicação de técnicas e critérios de teste, a fim de possibilitar o desenvolvimento de *softwares* de qualidade (COSTA JÚNIOR *et al.*, 2016).

1.2 Escopo do trabalho

O trabalho aborda a utilização de técnicas de V&V para a garantia da qualidade de *software* em um contexto real de aplicação, mostrando os benefícios que elas proporcionam para melhoria do processo de desenvolvimento e manutenção do sistema.

O escopo do projeto aborda como técnica de verificação estática as ferramentas automatizadas de inspeção do código-fonte e como técnicas dinâmicas o uso de testes unitários e testes de caixa-preta, apoiados por uma ferramenta desenvolvida pela própria equipe para melhorar a produtividade das tarefas de teste. Este trabalho não irá abordar todos os métodos de verificação de *software*, como por exemplo o teste de integração, mas somente as técnicas supracitadas, buscando respostas para as seguintes questões:

1. Qual o impacto da aplicação de técnicas de verificação estática automatizada do código-fonte para a qualidade do *software*?
2. Quais os benefícios da utilização dos testes unitários para o desenvolvimento e manutenção do sistema?
3. Como a utilização de ferramentas de testes de caixa-preta podem melhorar a cobertura dos

testes?

4. Como essas técnicas podem ser utilizadas em conjunto para obter uma melhoria na qualidade do *software*?

1.3 Objetivos

Este trabalho teve como finalidade realizar um estudo de caso das inspeções e testes em um sistema de roteirização de veículos, onde não se utiliza a maioria das técnicas de V&V no seu desenvolvimento. Para atingir esse objetivo o estudo buscou:

- Desenvolver uma ferramenta de teste de caixa-preta para validação;
- Desenvolver testes unitários para o SRV;
- Aplicar ferramentas de verificação estática automatizada de código-fonte;
- Aplicar a ferramenta de teste de caixa-preta desenvolvida para o SRV;
- Propor métricas para avaliar a eficiência das refatorações;
- Propor um processo de V&V;
- Analisar os resultados.

1.4 Metodologia e organização do trabalho

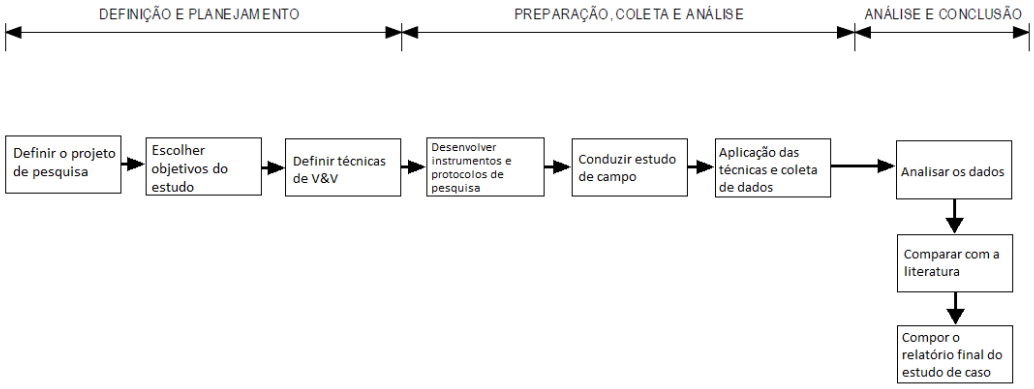
A metodologia de trabalho é estudo de caso e foi fundamentada em pesquisas bibliográficas, seguindo estratégias de investigação direta e intensiva, baseadas em observações empíricas. A pesquisa de campo foi realizada para levantar e analisar dados quantitativos e qualitativos através das técnicas de análise estática, testes unitários e testes de caixa-preta. A partir dos dados quantitativos gerados pela aplicação das técnicas, os aspectos particulares foram identificados e correlacionados para sintetizar uma relação entre eles e o salto na qualidade do *software*. Yin (2001) sugere a estrutura de projeto de pesquisa mostrado na Figura 1, sendo essa a sequência lógica que conecta os dados empíricos às questões de pesquisa iniciais do estudo e, em última análise, às suas conclusões.

Essa pesquisa conduz o estudo de um único caso em um SRV, de natureza quali-quantitativa e de finalidade descritiva, observando a aplicação das técnicas citadas no escopo do trabalho e quantificando os resultados. Com o estudo realizado por meio das observações empíricas, foi possível inferir no avanço da qualidade no que se refere a observações da equipe, e também foi possível chegar a conclusões através de comparações dos dados gerados pela

aplicação das técnicas em relação ao início e fim do projeto.

Após essa introdução, o trabalho está organizado como apresentado a seguir. No Capítulo 2 são discutido as técnicas de V&V e seus benefícios, bem como os trabalhos relacionados. A descrição do estudo de caso será apresentada em seguida no Capítulo 3. Por fim, os resultados e conclusões são apresentados no no Capítulo 4 e 5, consecutivamente.

Figura 1 – Metodologia do estudo de caso



Fonte: Adaptada de Yin (2001).

2 FUNDAMENTAÇÃO TEÓRICA

Segundo Pressman (2009), o processo de V&V inclue uma grande gama de atividades de Garantia da Qualidade de *Software* ou *Software Quality Assurance* (SQA): “revisões [...], análise de algoritmos, teste de desenvolvimento, teste de usabilidade, teste de qualificação, teste de aceitação e teste de instalação”.

De forma análoga Paula Filho (2009) afirma que, as verificações usadas nos processos de V&V incluem análises estáticas, testes de desenvolvimento (ou seja, de unidade e de integração) e revisões, ele considera ainda o teste de sistema como atividade de validação das conformidades do *software* com os requisitos. Este capítulo, foca principalmente nas inspeções de programa e nos testes em geral.

Um dos pioneiros da engenharia de *software*, Boehm (1979 apud SOMMERVILLE, 2010) define Validação e Verificação respectivamente como:

- Validação: “Estamos construindo o produto certo?”
- Verificação: “Estamos construindo o produto da maneira certa?”

A Verificação e Validação são coisas diferentes, embora sejam muito confundidas, essas, objetivam verificar se o *software* em desenvolvimento satisfaz suas especificações e oferece a funcionalidade esperada pelas pessoas que estão pagando pelo sistema. Esses processos iniciam-se assim que os requisitos estão disponíveis e continuam em todas as fases do processo de desenvolvimento (SOMMERVILLE, 2010). Para Terra e Bigonha (2008), verificação se destina a mostrar que o projeto do *software* atende a sua especificação, enquanto que a validação se destina a mostrar que o *software* realiza exatamente o que o usuário espera que ele faça.

2.1 Inspeções e testes de software

Inspeção é uma técnica de verificação estática, um dos principais benefícios é que pode ser aplicado a qualquer artefato produzido durante o desenvolvimento de *software*. Segundo Terra e Bigonha (2008), a análise estática de código é um dos instrumentos conhecidos pela Engenharia de Software para a mitigação de erros, seja por sua utilização para a verificação de estilos, para a verificação de erros ou ambos.

Mesmo com a utilização das inspeções de programa para manter os erros fora do *software* durante o desenvolvimento, o teste é um elemento crítico para a qualidade do produto, pois representa a revisão final da especificação, projeto e geração de código. Para Sommerville

(2007), mesmo que as inspeções de *software* sejam amplamente utilizadas, o teste de programa será sempre a técnica principal de verificação e validação do sistema.

Segundo Wilkerson *et al.* (2012), embora qualquer artefato de *software* possa ser inspecionado, a maior parte das pesquisas sobre inspeção aborda como foco a inspeção do código-fonte. Neste estudo, limitou-se as inspeções à análise estática com apoio de ferramentas e referindo-se a elas como inspeções de código.

De acordo com Medeiros (2017):

> A análise estática permite que diversos erros sejam encontrados antes mesmo que o programa tenha que ser compilado (somente válido para as ferramentas que analisam somente o código fonte). No entanto, é recomendado que ambas as análises sejam utilizadas em conjunto durante o desenvolvimento de um projeto. A análise estática pode encontrar problemas independentemente da entrada e da saída do programa, enquanto que a análise dinâmica pode encontrar problemas de codificação que não foram ainda informados como um padrão de erro nas ferramentas de análise estática.

O ideal é que as técnicas de teste sejam complementares, considerando a forma como são utilizadas, de modo que as a vantagens de cada uma delas possa ser bem explorada a fim de extrair o melhor de cada abordagem (COSTA JÚNIOR *et al.*, 2016).

Segundo Terra e Bigonha (2008):

> Em síntese, os testes de *software* e as ferramentas de verificação de erro detectam defeitos diferentes. Testes de *software* são eficazes em encontrar defeitos lógicos que são melhor visualizados quando o *software* está em execução, enquanto que as ferramentas de análise estática automatizada são eficazes em encontrar defeitos relacionados aos princípios de programação estruturada e à manutenibilidade. Portanto, para um bom projeto seria altamente recomendável a utilização de ambas as técnicas.

O teste de *software* é, de longe, o método mais comumente usado para garantia de qualidade e controle de qualidade em uma organização de desenvolvimento de *software*, e uma parte muito importante do processo de desenvolvimento (COLLINS *et al.*, 2012).

A importância e a complexidade do teste de *software* podem ser refletidas pelos custos envolvidos, onde, 30% a 80% dos custos de desenvolvimento estão relacionados aos testes (BOEHM, 1976; GAROUSI; ZHI, 2013; KARHU *et al.*, 2009), e os estudos sobre o tempo de liberação do *software* indicam que a maior parte dele é consumido por testes (KERZAZI; KHOMH, 2014). Seguindo essas afirmações, para Wiklund *et al.* (2017), o custo e o tempo gasto com testes podem ser gerenciados por meio da automação deles, em que a execução de um teste é realizada por um *software* em vez de uma pessoa.

2.2 Métricas de teste de software

Atividades de teste podem fornecer uma oportunidade crítica para capturar informações sobre métricas e defeitos que podem ser usadas para melhorar ambos os processos de desenvolvimento e teste, e ainda fornecer visibilidade na qualidade do produto e do processo (LAZIC; MASTORAKIS, 2008).

Lazic e Mastorakis (2008), apresenta algumas métricas que podem ser usadas para fornecer relatórios do *status* do projeto para o condutor de teste e gerente de projeto:

- Quantidade de casos de teste;
- Quantidade de casos de teste executados;
- Quantidade de casos de teste aprovados;
- Quantidade de casos de teste falhos;
- Tempo de execução do caso de teste;
- Tempo de execução do teste.
- % cobertura dos testes;

Para Lazic e Mastorakis (2008), essas métricas fornecem valorosas informações que quando usadas e interpretadas, conduzem para melhorias significantes de forma geral no ciclo de vida do desenvolvimento de *software*.

Hecht *et al.* (1977) propôs algumas métricas de qualidade para prontidão de teste, dentre elas estão as métricas listadas abaixo.

- Taxa de testes falhos: É a taxa de execuções de testes com falha do total de execuções de teste em um dado período de tempo, como uma medição e estimativa útil de confiabilidade de *software*. A taxa de falha foi considerada mais estável e, portanto, melhor que o número de testes falhos por tempo de execução. O estudo de Hecht *et al.* (1977), comparou a experiencia total do teste em um pacote de suporte de *software* para uma aplicação de transporte aéreo, quando relatou a taxa de testes falhos e testes falhos por tempo de execução acumulada sobre um período de tempo (BOWEN, 1979).
- % da cobertura de teste: Muitos contratos especificam que uma certa porcentagem de instruções deve ser executada com sucesso antes da aceitação do *software* pelo cliente. Ferramentas de análise estática e dinâmica de *software* estão disponíveis para coletar os dados necessários para calcular a porcentagem de cobertura de teste, essas ferramentas e métricas são adequadas, com exceção da cobertura de requisitos de desempenho, para o teste de qualidade de *software*.

Estão disponíveis ferramentas de teste de *software* que incorporam algoritmos e métricas de teste proprietários que podem ser usados para medir o desempenho e a conformidade do *software* (TASSEY, 2002).

2.3 Trabalhos relacionados

A fim de investigar inspeções e testes, estudos de casos são realizados e relatam experiências na indústria. Alguns destes têm um ponto de vista abrangente, embora não apresentem resultados estatisticamente garantidos. A observação dos experimentos pesquisados quando os estudos de casos apresentados avaliam o papel da inspeção versus teste ao encontrar falhas, leva em consideração também a inspeção de outros artefatos além do código-fonte. Assim, inspeções de documentos, como requisitos e *design* das especificações também estão em foco e a questão mais abordada nos estudos de casos que investigam inspeções e testes é uma simplificação de se vale a pena gastar esforços com as inspeções, em comparação com outras atividades de detecção de falhas. A opinião geral de vários desenvolvimentos diz que o uso de inspeções melhora a qualidade do produto e que reduzem o esforço de teste (AURUM *et al.*, 2002).

Em um estudo realizado por Rojas *et al.* (2016), foi investigado diferentes estratégias para geração de testes unitários, em particular técnicas de semeação de números e strings constantes derivados estaticamente e dinamicamente, semeação do tipo da informação e semeação de testes gerados previamente. Os resultados de uma análise empírica realizado em uma larga coleção de projetos de código aberto da SF110 corpus e do Repositório da Apache Commons são relatados. Esses experimentos mostram que, mesmo para uma ferramenta de teste capaz de conquistar alta cobertura, o uso de estratégias de semeação apropriada pode melhorar mais a performance.

Outro estudo realizado por Vonken e Zaidman (2012), investiga se a disponibilidade de testes unitários durante a refatoração realmente leva a refatorações mais rápidas e a códigos de alta qualidade após elas. Para isso, Vonken e Zaidman (2012) estabelece um experimento controlado em dois grupos envolvendo 42 participantes. Contudo, os resultados de Vonken e Zaidman (2012) indicam que a disponibilidade de testes unitários durante a refatoração não leva a uma refatoração mais rápida ou a um código de maior qualidade após a refatoração.

Já o trabalho de Larsen *et al.* (2005), apresenta uma ferramenta de teste caixa-preta baseada em modelos de testes de conformidade de sistemas embarcados de tempo real, chamada de UPPAAL-TRON. Em seu trabalho é apresentado a experiência em aplicar a ferramenta em

um estudo de caso industrial, concluindo que a ferramenta é aplicável a sistemas práticos, e que tem potencial promissor de detecção de erros.

Agora, um estudo voltado mais para inspeções que foi realizado por Misra *et al.* (2014), propõe um modelo para o processo de inspeção com a intenção de ser aplicável e aceito em pequenos e médios empreendimentos e grandes organizações de *software*. O modelo de Misra *et al.* (2014) foi implementado em duas organizações: um em uma companhia de médio porte e a outra em um departamento de uma grande companhia, onde a viabilidade e o beneficio da implementação foram confirmados. Misra *et al.* (2014) mostra também uma comparação realizada com modelos de inspeção alternativos recentes, mostrando a praticidade da proposta, facilidade de adoção e custo-efetividade.

Outro estudo voltado para inspeções feito por Araújo Filho *et al.* (2010), avalia o nível de correlação existente entre defeitos reportados por usuários finais (isso é, defeitos de campo) e *warnings* gerados pela ferramenta de análise estática *Findbugs*, largamente utilizada em sistemas Java. No estudo, procurou-se avaliar a existência de dois tipos de correlação: a direta (quando *warnings* podem contribuir para localizar e remover defeitos de campo) e a indireta (quando *warnings* são capazes de servir como indícios de futuros defeitos de campo). Como resultado, observou-se que não existe correlação direta entre defeitos de campo e *warnings*. No entanto, testes estatísticos mostraram que existe um nível significativo de correlação indireta entre *warnings* e tais tipos de defeitos.

Posteriormente, uma experimentação realizada por Wilkerson *et al.* (2012), comparou as taxas de defeitos de *software* e os custos de implementação de dois métodos de redução de defeitos: inspeção de código e desenvolvimento orientado a testes. Foram divididos participantes, consistindo de estudantes de informática júnior e sênior em uma grande universidade, em quatro grupos, e pediram que contemplassem a mesma atribuição de programação usando desenvolvimento orientado a testes, inspeção de código, ambos ou nenhum. Concluiu-se assim, com as contagens de defeitos e os custos de implementação resultantes entre os grupos, que a inspeção de código é mais eficaz do que o desenvolvimento orientado a testes na redução de defeitos, mas que também é mais cara, como também o desenvolvimento orientado a testes não era mais eficaz na redução de defeitos do que os métodos tradicionais de programação.

Inspeções e testes são as duas técnicas de verificação mais importantes das atividades de V&V, são técnicas que ambos os pesquisadores e profissionais precisam entender, sobre como usá-los separadamente, mas o mais importante, como combiná-las para alcançar maior eficácia.

Estudos empíricos comparando inspeções e testes, não dá uma resposta simples de como as técnicas estão relacionados e como combiná-los de forma eficiente, para isso é necessário que o estudo seja conduzido com o objetivo de usar as técnicas de forma que ambas se complementem.

A maioria dos experimentos controlados concentram nas inspeções e testes como atividades isoladas, e fazem comparações teóricas depois ou focam somente em uma das técnicas. Já os estudos de casos fazem tudo apontar para uma maior eficácia nas inspeções, em comparação com os testes, enquanto os experimentos não mostram qualquer evidência de que uma técnica específica é superior à outra. A maioria, embora não todos, dos diferentes estudos concluem que inspeções e testes são complementares, e a classificação das falhas não são as mesmas. Ambas as pesquisas, estratégias de estudos de casos e experiências, chegam à conclusão de que inspeções e testes detectam diferentes tipos de erros. Esses estudos muitas vezes levam a sugestões da aplicação das atividades em combinação, em vez de isolamento. No entanto, poucos estudos avaliam em detalhe as atividades em combinação, pois como combinar as atividades, e como ela vai afetar a qualidade do produto é bastante incerto. Assim, este conhecimento tem que ser construído, e de forma adequada estudos empíricos podem ser realizados através da aplicação de um estudo de caso.

Diferindo dos demais trabalhos pesquisados, este trabalho tem como objetivo realizar o estudo com foco na utilização das técnicas em conjunto para atingir um objetivo maior, que é a melhoria da qualidade do produto estudado, podendo inferir em um salto na qualidade do *software*.

3 ESTUDO DE CASO

Neste capitulo são apresentadas as ferramentas que foram utilizadas no estudo (seção 3.1), bem como as métricas utilizadas para análise dos resultados (seção 3.3.1) e o processo de verificação proposto (seção 3.2). Ainda nesse capitulo é apresentado o processo de coleta de dados (seção 3.3.1) e como esses dados serão medidos e avaliados.

3.1 Ferramentas utilizadas

Durante o planejamento e definição das técnicas de V&V que foram aplicadas ao sistema, foram também pesquisadas e definidas ferramentas para aplicação das técnicas definidas no escopo do estudo, a fim de realizar as atividades da forma mais automatizada possível, com propósito de agilizar a aplicação delas com um baixo impacto na produção da equipe de desenvolvedores. As subseções à seguir apresentam as ferramentas utilizadas para análise estática (seção 3.1.1), testes unitários (seção 3.1.2) e teste de caixa-preta (seção 3.1.3).

3.1.1 Ferramentas de análise estática

A análise estática pode ser realizada tanto de forma manual, através de *checklist*, quanto automatizadas, com apoio de ferramentas que realizam inspeções de código-fonte. Para este estudo de caso, foi definida como estratégia a utilização de ferramentas automatizadas como base para refatorações e melhorias no *software*. O Quadro 1 mostra as ferramentas pesquisadas que poderiam ser as possíveis soluções a serem utilizadas no projeto, bem como as ferramentas escolhidas que estão sublinhadas.

A escolha das ferramentas se deu tanto pela linguagem utilizada pelo sistema, como também por elas poderem ser utilizadas como *plugin* da plataforma de desenvolvimento Eclipse. Por mais que apresentem algumas verificações em comum, as ferramentas fornecem muitas outras que são distintas, dispondo da flexibilidade de configurar o que vai ser checado, e podendo até criar novas regras de verificações de acordo as necessidades do desenvolvedor.

As verificações feitas pelas ferramentas selecionadas se concentram em identificar estilos, boas práticas e *bugs*. Abaixo, estão descritas essas verificações e qual ferramenta será responsável por identificar cada tipo de problema que pode ser melhorados no SRV.

- **Verificação de estilos:** Considera elementos como identação, espaços e tabulações, convenção de nomes, número de parâmetros, alinhamento na vertical, formato e presença de

Quadro 1 – Ferramentas pesquisadas.

Nome	Objetivo
Sonar	Identificar maus cheiros, bugs e vulnerabilidades
Findbugs	Encontrar bugs e vulnerabilidades
Checkstyle	Verificar o cumprimento das regras de estilo
PMD	Identificar falhas comuns de programação
UCDetector	Identificar métodos e classes não utilizadas

Fonte: Elaborado pelo autor.

comentários, dentre outros. São todos os aspectos que contribuem para tornar o código mais padronizado, organizado e legível. A ferramenta utilizada para este tipo de verificação foi o Checkstyle;

- **Verificação de boas práticas:** Aplica várias regras para verificar se práticas corretas estão sendo realizadas, como evitar duplicação de código, tamanho de métodos e classes, tamanho de parâmetros, uso do padrão *Singleton*, criação desnecessária de variáveis locais e muitas outras. O conjunto de regras é extenso e visa garantir que o código apresente as melhores práticas possíveis. A ferramenta de verificação utilizada para identificar más práticas foi o PMD;
- **Verificação de *bugs* e vulnerabilidades:** Trata de encontrar erros e vulnerabilidades no sistema. Isto é importante para antecipar a identificação de problemas no *software* antes mesmo de entrar em produção. A ferramenta utilizada para identificação de bugs e vulnerabilidades foi o Findbugs.
- **Identificação de métodos e classes não utilizadas:** Trata de encontrar códigos que não estão sendo referenciados e não tem nenhuma utilidade para o sistema. Isto é importante para identificação de código morto no *software*, sendo que esses podem aumentar a complexidade de compreensão do sistema. A ferramenta utilizada para identificação de métodos e classe sem utilidades foi o UCDetector.

3.1.2 Ferramenta de geração de testes unitários

O desenvolvimento dos testes unitários foi de grande importância ao decorrer da aplicação das demais técnicas, pois teve o papel de preservar o funcionamento interno das unidades do sistema durante as mudanças realizadas no código-fonte. Para isso, com o sistema já desenvolvido e em produção, foi possível utilizar uma ferramenta que gera esses teste unitários de forma automática, não exigindo que a equipe desenvolva manualmente esses testes. A equipe foi responsável somente por corrigir alguns testes gerados pela ferramenta e desenvolver alguns

novos para complementá-los.

Após realizar pesquisas buscando pela ferramentas adequada para automatizar a criação dos testes unitários, foi encontrada e definida a ferramenta EvoSuite para este objetivo. Configurada como *plugin* da plataforma de desenvolvimento, ela por sua vez gera os testes de unidade para código Java. Segundo Fraser e Arcuri (2017), o EvoSuite foi considerada a ferramenta com maior pontuação da categoria.

Posteriormente à definição da ferramenta para geração dos testes unitário, foi definida como ferramenta para execução desses testes o JUnit, que é um *framework* que facilita o desenvolvimento e execução de testes unitários em código Java.

3.1.3 Ferramenta de teste de caixa-preta

A técnica de teste de caixa-preta tem o objetivo de avaliar o comportamento externo do componente de *software*, sem considerar seu comportamento interno. Os dados de entrada são fornecidos, o teste é executado e o resultado obtido é comparado com um resultado esperado previamente conhecido. Como detalhes de implementação não são considerados, os casos de teste são todos derivados da especificação (AMARAL *et al.*, 2010).

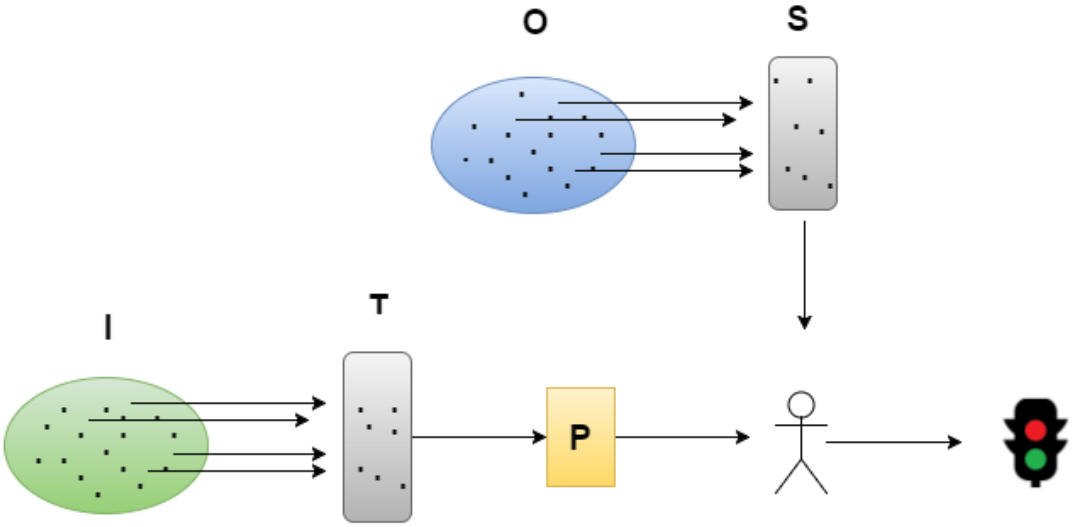
A ferramenta desenvolvida ajudou na elaboração de um padrão de validação, onde constantemente foi necessário validar o que foi desenvolvido ou refatorado. O teste de caixa-preta permitiu que as refatorações ocorressem de uma forma mais eficiente e rápida, possibilitando encontrar as não-conformidades do *software* em relação aos requisitos do sistema. Para realizar os testes, a ferramenta analisa as saídas do sistema a partir de determinadas entradas, identificando erros na saída para cada instância.

A Figura 2 ilustra como se comportam as atividades associadas ao teste. Costa Júnior *et al.* (2016) define que cada um dos itens apresentados na figura podem ser representados como:

- I: Domínio da entrada do programa P;
- T: Conjunto de dados de testes selecionados a partir de I;
- P: Programa em teste;
- O: Domínio de saída do programa P;
- S: Conjunto de saídas correspondentes às entradas T;
- Oráculo (ator): Representado pela figura do mecanismo que define se as saídas dos testes estão de acordo com o esperado.

De acordo com a Figura 2, um domínio de entradas I é definido e a partir dele é

Figura 2 – Atividades do teste de software.



Fonte: Costa Júnior *et al.* (2016).

selecionado um conjunto de dados T que representam as instâncias de entrada para o programa P, que a partir delas, irá gerar suas respectivas saídas. Além disso, um domínio de saída O é definido e o conjunto de saída S correspondentes as instâncias de entrada T são selecionados a partir de O. Com isso, o oráculo, que nesse caso é representado pela ferramenta proposta, define se as saídas estão de acordo com o esperado.

A princípio, no planejamento da ferramenta, foram analisadas e identificadas as possíveis verificações que deveriam ser implementadas para validar as saídas do SRV, com base nas análises realizada foram definidas as verificações apresentadas abaixo:

- Quantidade de clientes;
- Tempo de rota e quilometragem observada;
- Quebra de Tempo Máximo de Entrega(TME);
- Atendimento de janelas;
- Peso total da rota;
- Início e fim de rota devem ser iguais;
- Unicidade de clientes;
- Verificar contadores no resumo do grupo com almoço;
- Verificar contadores no resumo do grupo com pernoite;
- Rotas erradas;
- Semanas obrigatórias respeitadas;
- Dias obrigatórios respeitados;
- Frequência de visitas;

- Clientes com vendedores fixos;
- Dias não planejados;
- Verificar rotas que não deveriam ser euclidianas.

O Analisador é uma aplicação *Desktop* desenvolvida em *Java* que verifica se o resultado de um serviço *HTTP* do SRV para uma ou várias instâncias de entrada está correto. Para executá-lo é necessário informar a *URL HTTP* do serviço, diretório das instâncias de entrada, diretório onde serão armazenadas as saídas, e quais instâncias do diretório de entrada serão enviadas ao SRV. Posteriormente à execução da ferramenta, todas as entradas serão executadas e o Analisador irá dar uma saída para cada instância com os resultados, dessa forma será possível o testador analisar se o sistema teve um comportamento inesperado para determinada entrada ou se a saída é válida.

3.2 Definição do processo de verificação e validação

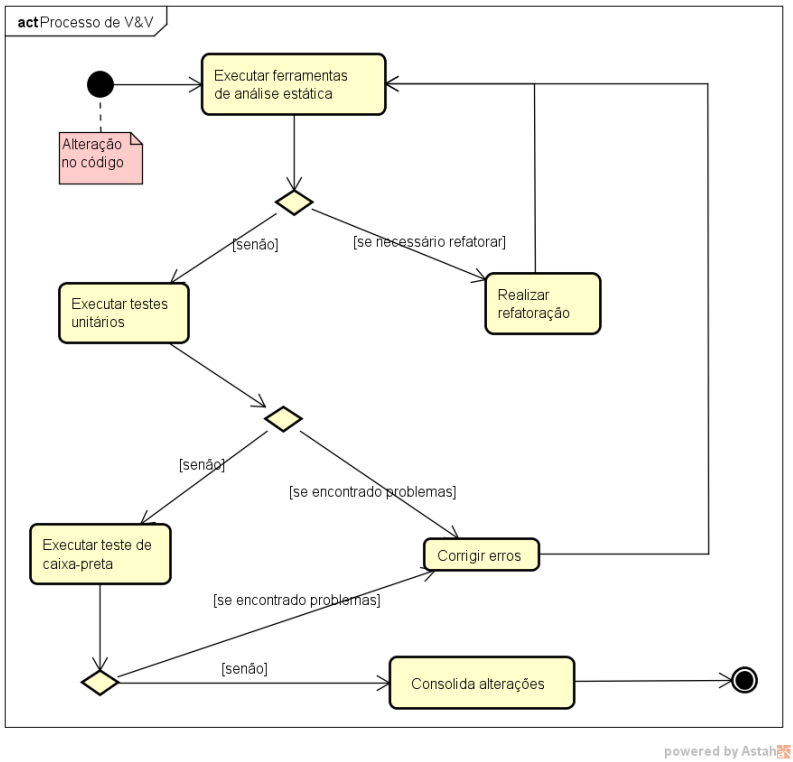
Para executar as alterações no código-fonte sem que seja inserido novos erros ou mesmo seja alterado o comportamento esperado do sistema, foi definido um processo de verificação, validação e correção de erros, utilizado para verificar o comportamento após as mudanças, através das ferramentas, dos testes unitários e também validá-las através do teste de caixa-preta. Esse processo busca combinar a utilização das técnicas em conjunto para aumentar a cobertura das verificações. Assim, sabendo que, inspeções e testes têm, cada um, vantagens e desvantagens que devem ser utilizadas em conjunto no processo de V&V (TERRA; BIGONHA, 2008). A Figura 3 apresenta o processo proposto.

De acordo com Terra e Bigonha (2008):

> O processo de V&V possui duas abordagens complementares para a verificação e análise do sistema que são os testes de *software* e as inspeções de *software*. O teste de *software*, que é uma técnica dinâmica, é a principal e mais utilizada técnica de V&V. Contudo, a inspeção de *software* vem sendo largamente utilizada pelo simples fato de as pesquisas demonstrarem que os defeitos encontrados por ela são completamente diferentes daqueles encontrados pelo teste de *software*. Logo, é recomendada a utilização destas técnicas em conjunto no processo de V&V.

Quanto a ordem das técnicas, segundo Medeiros (2017), é recomendado que as ferramentas de análise estática sejam aplicadas nos estágios iniciais da implementação, pois dessa forma a etapa de validação será menos custosa, isso porque as ferramentas poderão encontrar bugs, problemas de segurança e de performance antes mesmo da fase de validação. Já os testes automatizados devem começar pelos testes de unidade, que são pequenos trechos como

Figura 3 – Verificação e validação de alterações.



Fonte: Elaborada pelo autor.

funções, métodos ou classes, podendo ser realizados pelas ferramentas chamadas de arcabouços, como o JUnit, JSUnit, CSUnit e etc (AMARAL *et al.*, 2010).

De acordo com Louridas (2006):

> Programadores geralmente empregam verificadores estáticos após a compilação e antes dos testes. Deste modo, eles trabalham com um programa que tem uma indicação inicial de correção (pois ele compila) e tenta evitar deslizes e perigos bem conhecidos antes de batê-lo contra sua especificação (quando ele é testado).

Segundo Sommerville (2007), após um defeito ter sido descoberto, você precisa corrigi-lo e revalidar o sistema. Isso pode envolver uma nova inspeção do programa ou o teste de regressão nos quais os testes existentes são executados novamente (TERRA; BIGONHA, 2008). Pode-se assim, começar o processo de V&V do sistema com inspeções no início, mas, uma vez que um sistema esteja integrado, é preciso testá-lo para verificar as propriedades emergentes e se a funcionalidade do sistema é a que seu proprietário realmente deseja (SOMMERVILLE, 2007). Com isso, o processo apresentado na Figura 3 buscar realizar as verificações a cada alteração e/ou correção no *software* seguindo a ordem de aplicação das técnicas, começando com as ferramentas de análise estática, seguidas pelos testes unitários e por fim a validação com o teste de caixa-preta.

3.3 Coleta de dados

Para coletar os dados qualitativos, foram utilizadas técnicas de observação direta, como também o registro de dados quantitativos obtidos a partir da aplicação das ferramentas. Esses dados, tanto os qualitativos quanto os quantitativos, são de grande importância para analisar se os objetivos do projeto foram atingidos.

3.3.1 Dados quantitativos

Os dados quantitativos foram coletados a partir da aplicação das ferramentas de análise estática de código-fonte como também do JUnit. Essas ferramentas fornecem dados, como a quantidade de erros encontrados no projeto, ou mesmo dados mais precisos, como uma análise de erros por KLOC (mil linhas de código), erros por projeto ou erros por métodos, como é o caso da ferramenta PMD.

Através desses dados que são gerados pelas ferramentas, foi possível definir métricas para medir a qualidade do *software*, como também identificar indicadores de melhoria, através destas que estão listadas abaixo:

- *Quantidade de Defeitos Encontradas no Produto* (QDP): A quantidade de defeitos de um sistema em produção é uma métrica muito importante para mostrar a efetividade das tarefas de teste e correções de erros;
- *Tipos de Defeitos Encontrados* (TDE): Para aumentar a efetividade dos testes é importante saber os tipos e quantidade de erros que podem ser encontrados no sistema que está sendo testado. Essas informações podem ser historicamente úteis para fazer previsões quanto à qualidade do *software*. Os tipos de defeitos são diversos, podendo variar de erros simples de estilos e más práticas a erros de semântica, entre outros;
- *Defeitos por KLOC* (DKLOC): É o número de defeitos encontrados a cada mil linhas de código. Indica a qualidade do produto testado. Essa métrica é calculada a partir da relação entre o número de defeitos encontrados e o número de linhas de código;
- *Defeitos por Método* (DM): É o número de defeitos encontrados a cada método. Indica a qualidade do produto a partir do calculo da relação do número de defeitos encontrados e o número de métodos do sistema;
- *Defect Reduction Efficiency ou Eficiência de Redução de Defeitos* (DRDE): A eficiência de redução de defeitos é uma métrica orientada a função, proposta nesse trabalho para

fornecer uma medida da capacidade da equipe de reduzir defeitos. É calculado a partir da quantidade de defeitos reduzidos pela equipe dividido pelo número de defeitos encontrados na primeira medição, resultando em um percentual onde quanto mais alto maior a eficiência. A fórmula é dada por:

  - QDP_0: Valor da métrica QDP na primeira medição, quantidade inicial de erros encontrados no produto;
  - QDP: Valor da métrica QDP na medição atual, quantidade de erros encontrados no produto no momento da medição.

$$DRDE = \begin{cases} 0, & \text{se } QDP_0 - QDP \leq 0 \\ \frac{QDP_0 - QDP}{QDP_0}, & \text{senão} \end{cases}$$

- *Dynamic Defect Removal Efficiency ou Eficiência Dinâmica de Remoção de Defeitos* (DDRE): Essa é uma métrica orientada a função, proposta nesse trabalho como uma adaptação do *Defect Removal Efficiency ou Eficiência de Remoção de Defeitos* (DRE) para fornecer uma medida da capacidade da equipe de remover defeitos, mas incluindo os novos defeitos que foram inseridos e resolvidos ao longo das refatorações. É calculado a partir da relação do somatório de defeitos resolvidos pela equipe desde a primeira medição dividido pelo somatório de defeitos encontrados em todos os períodos até o momento. A fórmula dessa métrica é dada por:
  - P: Período analisado (onde são realizadas as medições);
  - r_p: Quantidade de erros resolvidos em determinado em um período p;
  - i_p: Quantidade de novos erros que foram identificados em um período p;
  - QDP_0: Valor da métrica QDP na primeira medição, quantidade inicial de erros encontrados no produto.

$$DDRE = \frac{\sum_{p=1}^{P} r_p}{QDP_0 + \sum_{p=1}^{P} i_P}$$

Já para os testes unitários, foram contabilizados os erros identificados durantes vários períodos através do número de falhas apresentados pela ferramenta JUnit. Foram utilizadas duas das métricas supracitadas para avaliar a identificação e remoção de erros detectados pelos testes unitários, a QDP e DRDE, como também, a métrica abaixo:

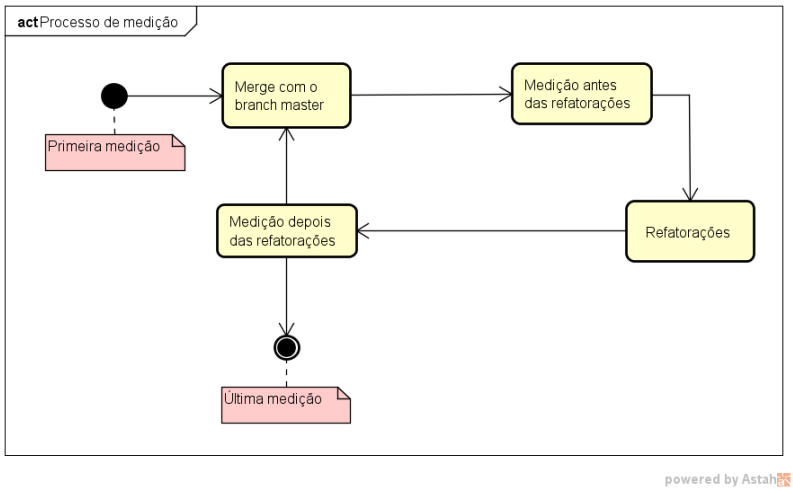
- *Erros por caso de teste* (ECT): É o número de erros identificados para cada caso de teste. É calculado a partir da quantidade total de falhas identificadas pelos testes unitários dividida pela quantidade de casos de teste desenvolvidos para o sistema. A fórmula é dada por:

– QCT: Quantidade total de casos de teste desenvolvidos e executados.

$$ECT = \frac{QDP}{QCT}$$

Para utilização dessas métricas, foi criado um processo para coleta de dados quantitativos gerados pelas ferramentas. Esse processo tem o objetivo de registrar todas as informações geradas pelas ferramentas, capturando o estado do sistema antes e depois de cada conjunto de refatorações/correções indicada por elas. O processes de coleta de dados é apresentado na Figura 4.

Figura 4 – Processo de medição dos indicadores.



Fonte: Elaborada pelo autor.

Antes das primeiras refatorações do código-fonte, foram aplicadas as ferramentas e registrados todos os dados resultantes dessa aplicação sobre os erros que foram identificados no sistema. Após isso, foram feitas as refatorações e registros do progresso em mitigar os erros identificados pelas ferramentas sempre antes e depois de cada ciclo de refatorações. Todas as refatorações foram realizadas em um *branch* criado especificamente para isso, sendo esse uma cópia do repositório principal onde as alterações desenvolvidas pela equipe são incorporadas. Assim antes de cada conjunto de refatorações era feito um *merge* do repositório principal para o *branch* das refatorações, incorporando as novas mudanças e logo após realizando um novo registro das métricas. Assim foi possível comparar o sistema no início e fim do estudo, relacionando a diminuição desses erros através das métricas definidas a melhoria na qualidade do *software*.

4 RESULTADOS

Os resultados obtidos seguiram os passos do processo modelado na Seção 3.2 para aplicação das técnicas e ferramentas escolhidas. Assim, o processo seguiu de forma sistemática garantindo que as refatorações indicadas pelas ferramentas e as novas modificações pudessem ser consolidadas após os erros serem corrigidos.

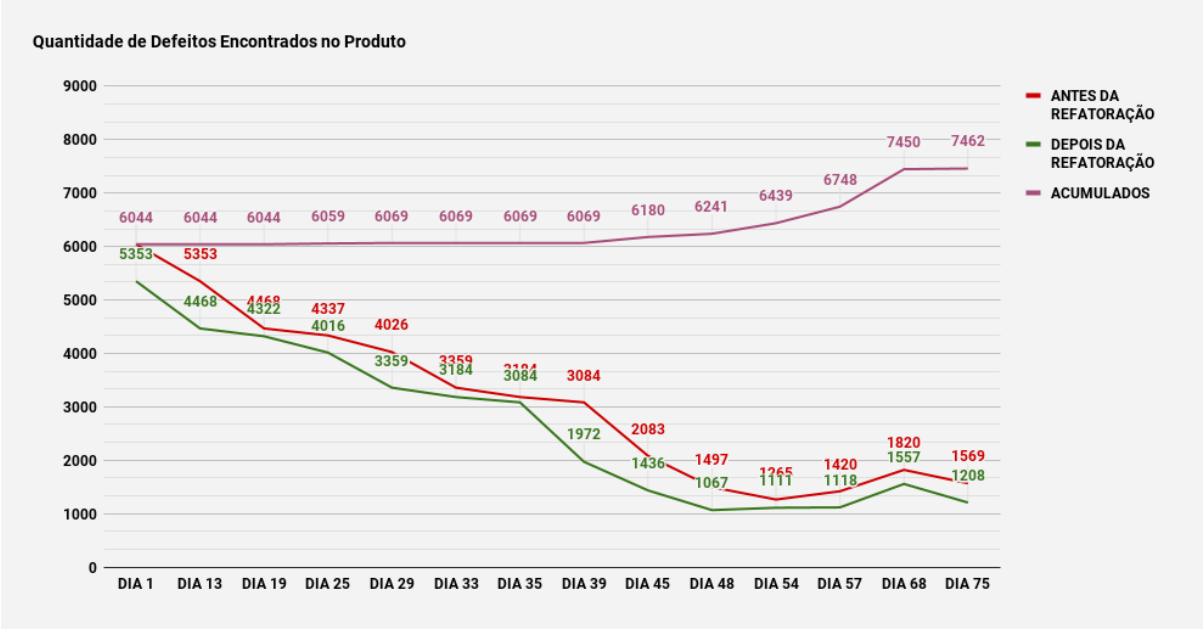
4.1 Análise dos dados coletados pelas ferramentas de análise estática

Para analisar os dados quantitativos coletados a partir dos erros identificados pelas ferramentas de análise estática, é preciso levar em consideração o declínio de alguns indicadores e o aumento de outros que estão relacionados com as métricas definidas na Seção 3.3.1, a cada medição realizada ao longo das refatorações feitas no SRV.

4.1.1 Quantidade de defeitos encontrados no produto

Essa métrica foi de grande importância para avaliar a quantidade total de erros em que o sistema possuía a cada medição, como também para verificar a eficiência na remoção desses defeitos. A Figura 5 mostra um gráfico, onde é possível verificar a redução da quantidade de defeitos do sistema à medida que as refatorações evoluíam e novas medições eram realizadas, mostrando o estado do sistema antes e depois de cada refatoração.

Figura 5 – Quantidade de defeitos encontrados no produto.



Fonte: Elaborada pelo autor.

Através do gráfico, é possível ver como a quantidade de erros identificadas declina a cada conjunto de refatorações, na respectiva data. É possível ver também que em algumas datas a quantidade de erros aumenta na medição realizada antes das refatorações, quando comparada com o estado do indicador na data anterior. Isso é decorrente dos *merges* feitos com o repositório principal do sistema, onde constantemente são inseridos novos códigos e consequentemente novos erros. A partir disso, foi possível realizar também a medição de quantos erros seriam acumulados caso as refatorações não estivessem sendo feitas, onde em um período de pouco mais que dois meses, aproximadamente 1400 novos erros foram inseridos no sistema junto com novos incrementos no código.

Analisando o gráfico e comparando o inicio e fim das refatorações é possível notar a diferença na quantidade de erros identificados, onde no inicio eram 6044 e na última medição estava com 1208 erros, um valor muito inferior quando comparado com o estado inicial do sistema na primeira medição.

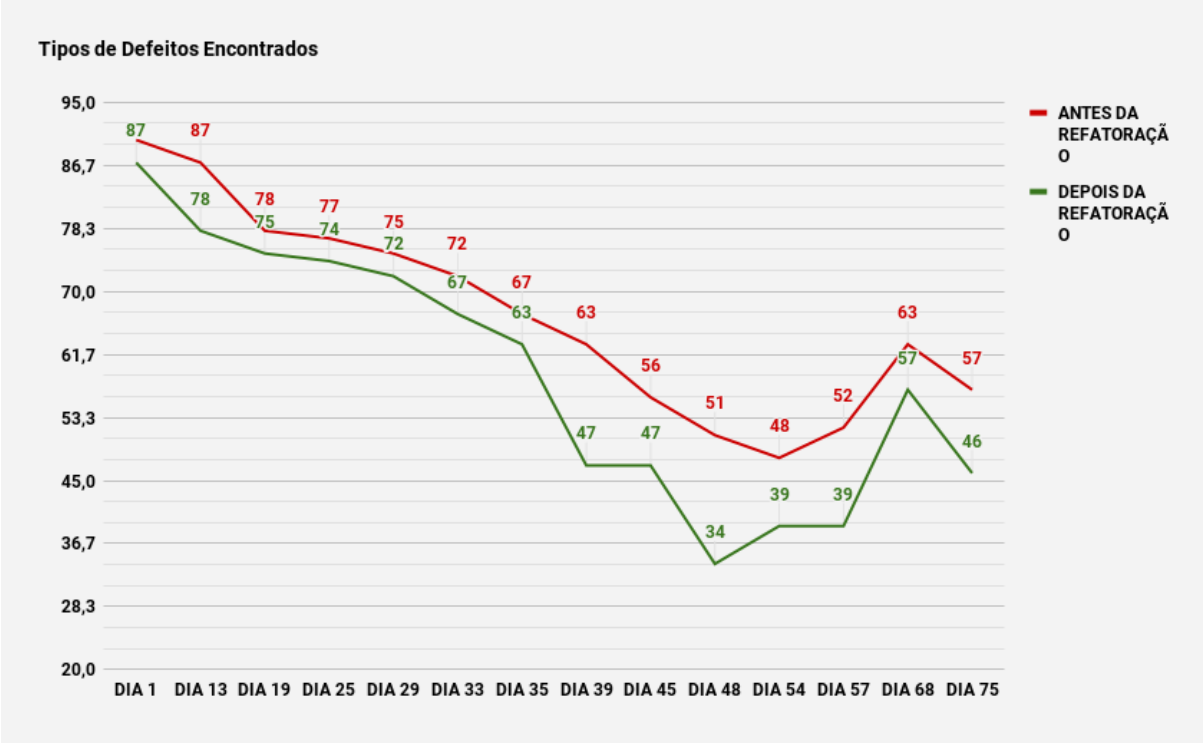
4.1.2 Tipos de defeitos encontrados

Durante a aplicação das ferramentas foram identificados erros de diversos tipos, dentre eles, erros de padrões de nomenclatura, complexidade ciclomática, código morto, más práticas de programação, entre outros. A partir dos defeitos encontrados no sistema, a métrica TDE tem o objetivo de quantificar os diferentes tipos de erros identificados. Com isso, foi possível visualizar a diminuição da quantidade de tipos diferentes de erros diminuir à medida que outros indicadores também declinavam, mantendo ainda um padrão no comportamento do gráfico mostrado na Figura 6 em relação ao comportamento de outros indicadores.

Assim como no gráfico apresentado na Figura 5 esse tem um comportamento parecido, pela relação entre a diminuição da quantidade de defeitos com a diminuição dos diferentes tipos, como também apresenta dados do antes e depois das refatorações, mostrando que em alguns momentos antes das refatorações tinha um aumento em relação ao estado anterior em que o sistema se encontrava, possibilitando ver que tanto alguns tipos de defeitos que já tinham sido eliminados voltassem, após novos incrementos no código, como também o aparecimento de novos tipos de erros.

Ao realizar a análise do gráfico é possível comparar o antes e depois do processo que o sistema foi submetido. Assim, podemos ver uma quantidade de tipos diferentes de erros inferior depois das refatorações, onde na primeira medição eram 90 tipos e ao final somente

Figura 6 – Quantidade de tipos de defeitos encontrados.



Fonte: Elaborada pelo autor.

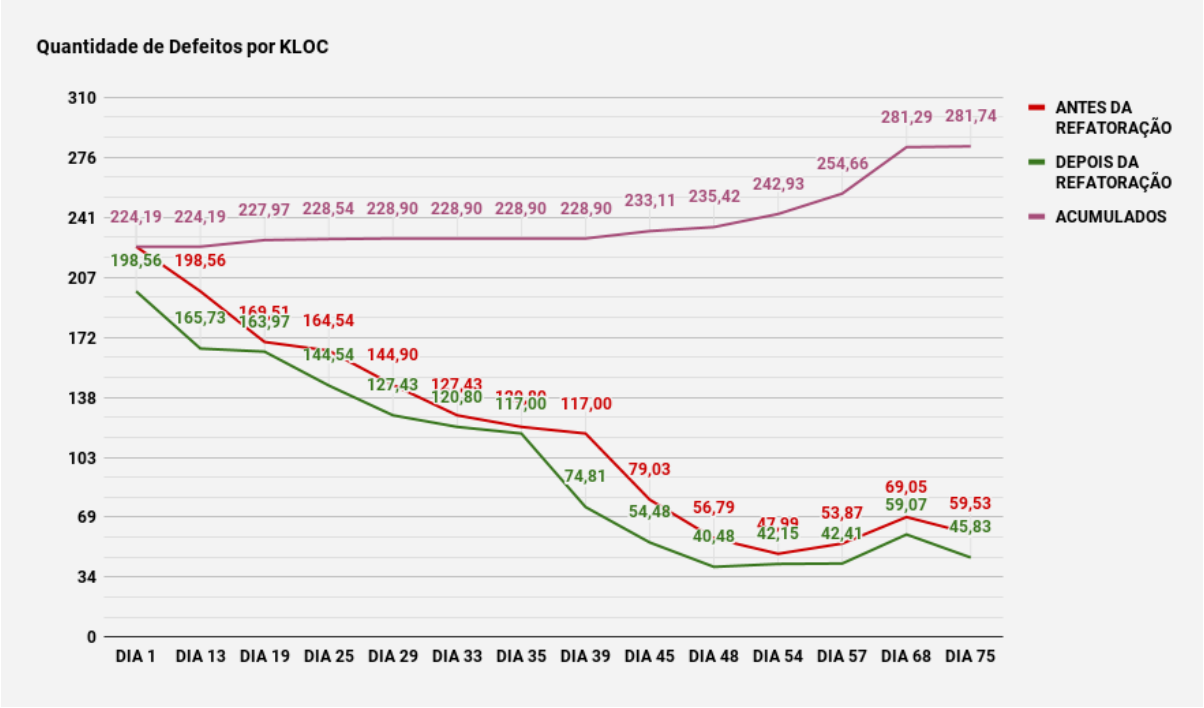
46, onde esses que restaram são erros que foram avaliados e menos priorizados em relação aos outros que foram corrigidos.

4.1.3 Defeitos por KLOC e Defeitos por método

O número de defeitos por cada mil linhas de código e defeitos por método são métricas de produto de *software* que tiveram grande importância para medir a qualidade do sistema. A partir delas foi possível notar diferenças na quantidade de defeitos que o SRV possuía antes e depois do processo de verificação, validação e refatorações. A Figura 7 mostra o gráfico da quantidade de defeitos por KLOC e em seguida a Figura 8 mostra o da quantidade de defeitos por método do sistema.

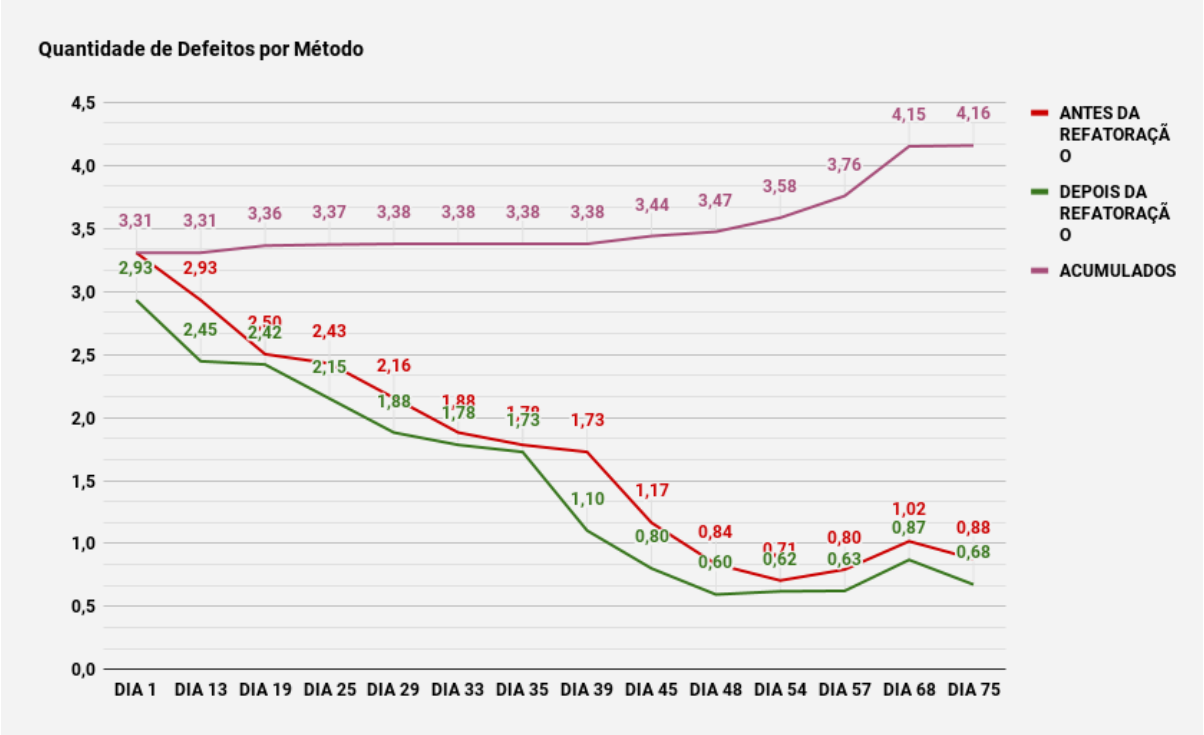
Realizando uma análise dos dois gráficos, é fácil ver que as linhas do antes, depois e acumulado são muito parecidas. Isso acontece por conta da relação que uma métrica tem com a outra, pois ambas estão relacionadas com o tamanho do *software*. Assim, é possível notar também que houve um declínio considerável dos indicadores, podendo inferir em um aumento de qualidade com a diminuição dos erros. A partir dos dois gráficos, nota-se que antes das refatorações a quantidade de DKLOC eram 224,19 e de DM 3,31, enquanto na última medição estavam com 45,83 DKLOC e 0,68 DM. Contudo, ainda é possível observar que se as

Figura 7 – Quantidade de defeitos por KLOC.



Fonte: Elaborada pelo autor.

Figura 8 – Quantidade de defeitos por método.



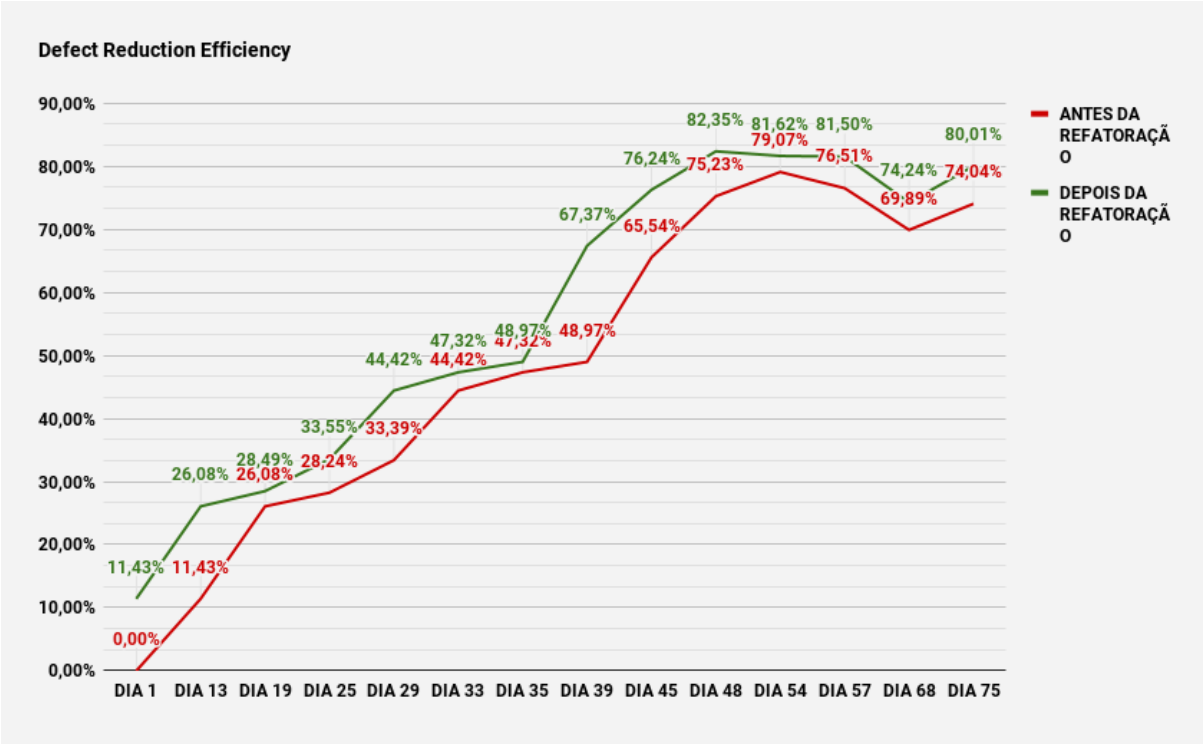
Fonte: Elaborada pelo autor.

refatorações não estivessem sendo feitas o sistema teria alcançado 281,74 DKLOC e 4,16 DM.

4.1.4 Eficiência de redução de defeitos

Diferente das outras métricas que foram utilizadas, a DRDE tem a característica de aumentar à medida que os erros são removidos, essa que é uma métrica orientada a função pode indicar o quão eficiente é o esforço da equipe para reduzir os defeitos. Com isso, essa é uma das principais métricas avaliadas nesse trabalho, pois mede a efetividade das refatorações em reduzir os erros e aumentar a qualidade. A Figura 9 mostra o aumento da DRDE no gráfico ao passo que os defeitos identificados eram corrigidos.

Figura 9 – Eficiência de redução de defeitos.



Fonte: Elaborada pelo autor.

Através do gráfico da DRDE é possível perceber o aumento significativo da efetividade de correções de erros, partindo de 0,00% de eficiência para 80,01%. Contudo, é notável que a DRDE diminuía e aumentava à medida que novos erros eram inseridos no código-fonte, sendo possível ver que os valores tendem a estabilizar quanto mais próximo de zero o número de erros chega. Com isso, é importante ressaltar que o processo de verificação, validação e refatoração deve ser contínuo para manter a qualidade do *software* em um nível satisfatório.

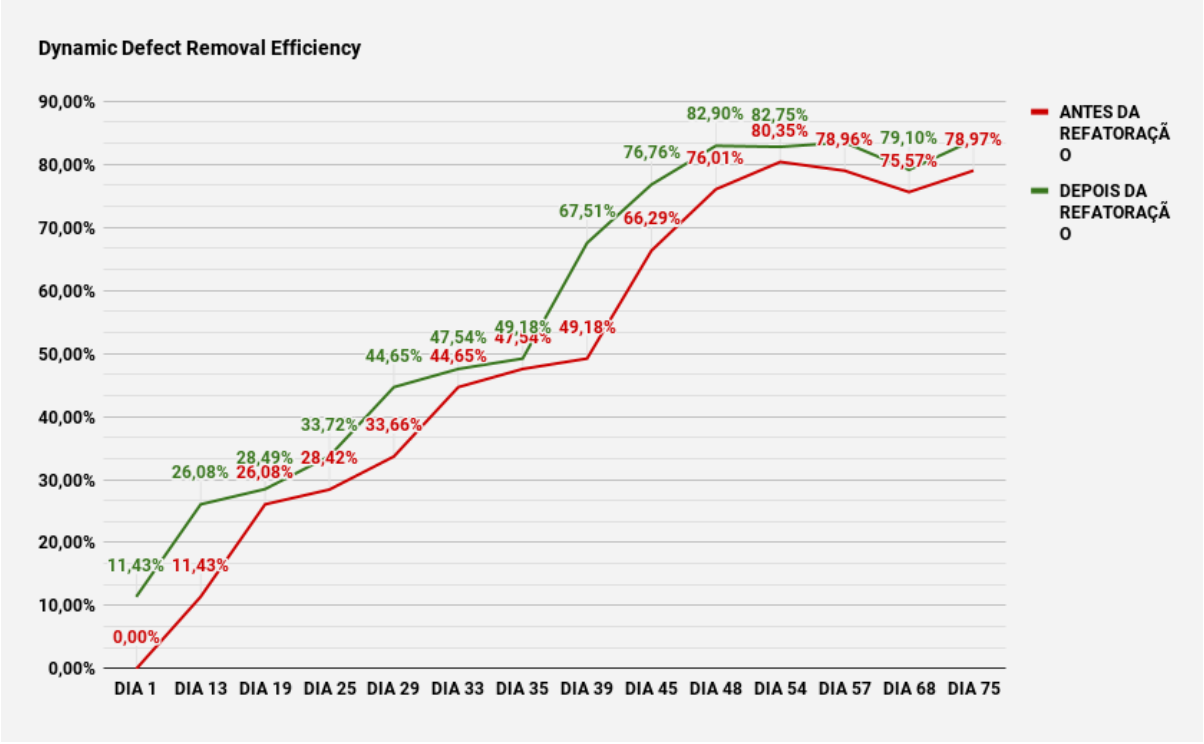
Esses valores foram calculados levando em consideração a quantidade de erros encontrados na primeira medição e também através da quantidade de erros restantes. Assim, a quantidade de erros da primeira medição menos os erros restante, resultando em um valor que

representa quantos erros foram reduzidos desde o inicio das refatorações.

4.1.5 Eficiência dinâmica de remoção de defeitos

Essa métrica foi utilizada para medir a eficiência de remoção de defeitos considerando aqueles que aumentavam e diminuíam dinamicamente com as alterações constantes no código-fonte do sistema. Com a DDRE, é possível ver o esforço da equipe em diminuir a quantidade de defeitos no *software*, mesmo com novos erros sendo inseridos a todo momento. Assim, mesmo quando uma grande quantidade de defeitos estão sendo inseridos no código e a equipe consegue reduzir eles para o mesmo valor ou um maior que a medição da última refatoração feita, a DDRE ainda vai ser mais alto, pois, a quantidade de erros é quase a mesma, mas a quantidade de defeitos resolvidos é maior. A Figura 10 mostra o gráfico da DDRE com os valores calculados em cada período.

Figura 10 – Eficiência dinâmica de remoção de defeitos.



Fonte: Elaborada pelo autor.

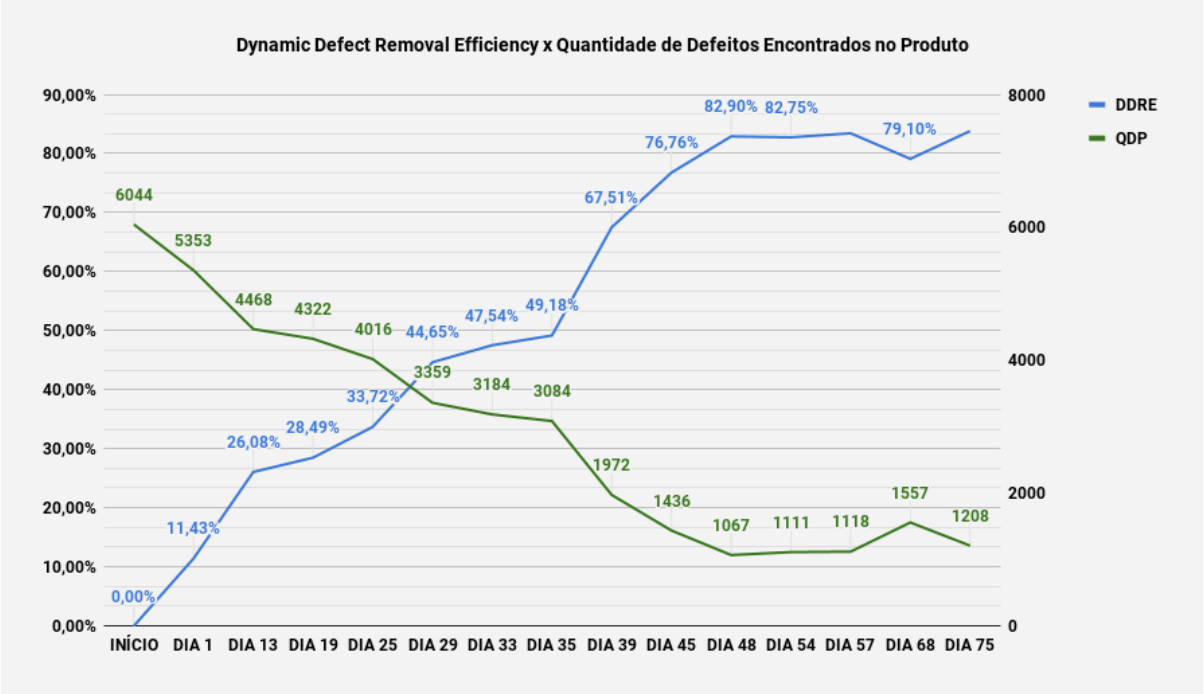
Através do gráfico da DDRE é possível perceber um aumento significativo da efetividade de correções de erros, partindo de 0,00% de eficiência para 83,81%. Esses valores foram calculados levando em consideração a quantidade total de erros encontrados e também através da quantidade total de erros resolvidos até o momento da medição. Assim, a quantidade

total de erros encontrados até um período menos a quantidade de erros restantes, resultando em um valor que representa o total de erros removidos até o momento, incluindo os que foram introduzidos em novos incrementos no código depois da primeira medição.

4.1.6 Eficiência dinâmica de remoção de defeitos x Quantidade de defeitos encontrados no produto

Realizando uma análise sobre as métricas utilizadas no estudo, duas dessas tem uma relação forte no comportamento dos indicadores de qualidade, a DDRE e a QDP. Assim, é possível ver essa relação no gráfico apresentado na Figura 11.

Figura 11 – Eficiência dinâmica de remoção de defeitos x quantidade de defeitos encontrados no produto.



Fonte: Elaborada pelo autor.

A partir do gráfico é fácil ver que quanto mais a quantidade de defeitos diminui, maior é a DDRE, de forma análoga, quanto mais eficiente for o esforço da equipe para remover os defeitos identificados, menos erros vão ser encontrados no produto. Contudo, mesmo que na teoria a QDP ideal seja zero, na prática é muito difícil isso acontecer, principalmente quando não existe um processo para garantir que esses defeitos sejam corrigidos antes da alteração ser consolidada e incorporada ao repositório do sistema.

Após visualizar as últimas datas de refatorações, percebe-se que quanto mais a QDP diminui, há uma tendência da quantidade de defeitos manter-se baixa, mas sem grandes variações.

Isso acontece pelo fato de que os erros restantes possuem uma complexidade maior de resolução, tornando mais difícil eliminar todos os defeitos com novos sendo inseridos constantemente. Para isso, seria necessário que os erros fossem eliminados antes mesmo do código que estão inseridos ser incorporado ao repositório do *software*.

Assim, tanto nas correções de problemas no sistema quanto no desenvolvimento de novas funcionalidades, pode ser utilizado um processo como o que foi proposto na Seção 3.2, fazendo com que toda alteração no código-fonte tivesse que ser verificada e validada antes de se consolidar. Com isso, toda alteração feita no sistema já seria incorporada sem os erros que foram verificados pelo próprio desenvolvedor, diminuindo o esforço de refatorações com o proposito de remover defeitos que já estão inseridos no produto.

4.2 Ferramenta de teste funcional

A ferramenta planejada e descrita na Seção 3.1.3 foi desenvolvida e utilizada como forma de validação no processo apresentado na Seção 3.2. O Analisador encontra-se na sua terceira versão após refinamentos decorridos de *feedbacks* da equipe de desenvolvedores do sistema.

A ferramenta foi muito importante no processo de correção dos erros verificados, pois através dela foi possível identificar erros nas saídas, logo após algumas refatorações terem inserido problemas na lógica da aplicação, resultando em um comportamento inesperado. Assim quando eram feitas alterações no sistema, essas só eram validadas após passar por todo o processo proposto.

Durante as modificações a que o sistema foi submetido, foi possível fazer comparações entre *branchs* do sistema, como por exemplo a execução do teste de caixa-preta no *branch master* e no *branch* refatorações, esse que é onde a mudança no código foi realizada. Com isso, foi possível avaliar se uma refatoração que deveria manter o mesmo comportamento após uma melhoria no código-fonte teria inserido algum erro. Assim, com o *branch* refatorações acompanhando sempre o desenvolvimento do *master* era esperado que os dois tivessem o mesmo comportamento. A Figura 12 mostra a saída de uma instância executada no repositório *master*.

A Figura 12 mostra o resultado de uma execução de sucesso realizada no código-fonte do repositório *master* e que quando executado para o *branch* refatorações deve retornar a mesma saída. Essa figura mostra um teste realizado com o proposito de validar uma mudança realizada no código-fonte do repositório refatorações onde foi identificado um erro ao executar

Figura 12 – Saída de instância executada no *master*.

GRUPOS

Num. de grupos: 3

ID Grupo	Peso ocupado	Tempo ocupado	Num. de visitas feitas	Cubagem ocupada	Núm. de janelas quebradas	Placa do veículo
0	10750	8,905	16	0	0	
1	11250	10,781	17	0	0	
2	4500	2,343	2	0	0	

VEÍCULOS

Quantidade de veículos usados: 3

Quantidade de veículos ociosos: 2

ID Grupo	Placa	Peso limite	Peso ocupado	Tempo limite de serviço	Tempo serviço ocupado	Núm. limite de visit	Visitas feitas	Cubagem limite	Cubagem ocupada
0		11300	10750	12	8,905	45	16	54	0
1		11300	11250	12	10,781	45	17	54	0
		10000	0	12	0	45	0	48	0
		6800	0	12	0	45	0	32,4	0
2		6800	4500	12	2,343	45	2	32,4	0

CLIENTES ATENDIDOS

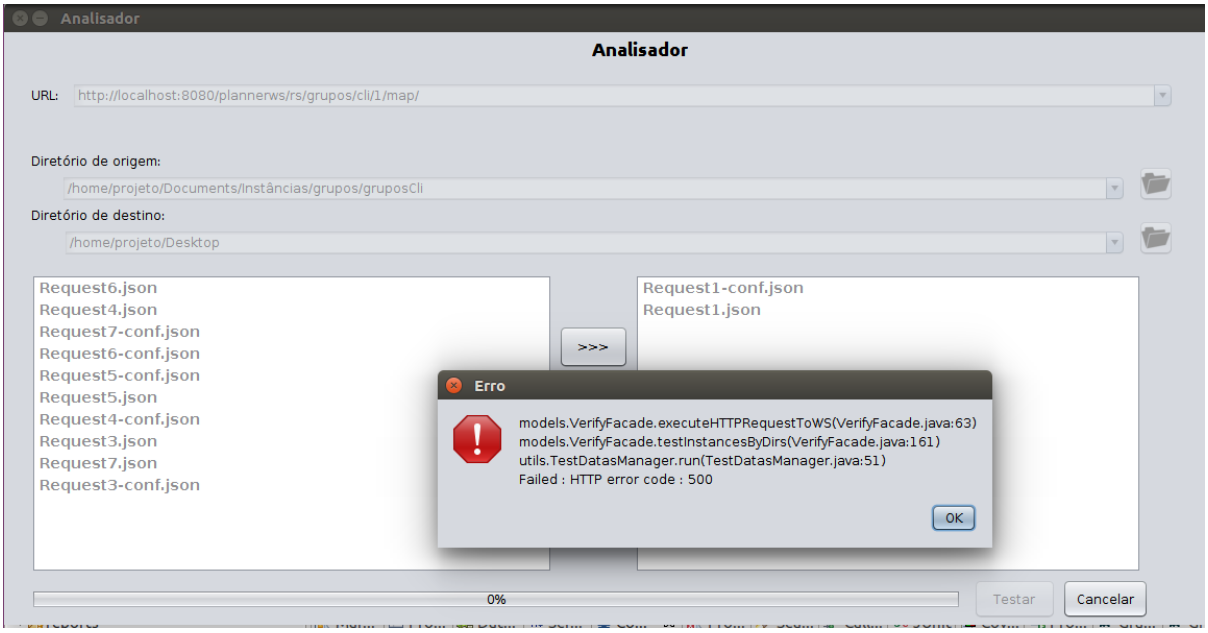
Quantidade: 35

ID Grupo	ID	X	Y	Demanda de peso	Tempo de serviço	Núm. visitas limite	Kilometragem atua	xpediente de iníci	ní-cio atendiment	Expediente de fim	Fim atendimento
0	1	-39,27314	-7,299976	0	0	0	153,764	6	6	18	14,905
0	4	-38,75605	-7,413934	0	0,0833333333	0	0	6	6	18	6,2
0	14	-38,77452	-7,386769	0	0,0833333333	0	5,11	6	6,29	18	6,57
0	35	-38,77902	-7,387323	0	0,0833333333	0	5,815	6	6,59	18	7,27
0	18	-38,84219	-7,313114	0	0,0833333333	0	20,976	6	7,48	18	7,65
0	27	-39,23279	-7,264703	0	0,0833333333	0	82,206	6	8,52	18	8,8
0	25	-39,2776	-7,282155	0	0,0833333333	0	89,687	6	8,93	18	9,09

Fonte: Elaborada pelo autor.

sendo que para o *master* realizou a execução com sucesso. A Figura 13 mostra o resultado apresentado pela ferramenta de teste de caixa-preta quando executada para o *branch* refatorações com a mesma instância que foi apresentada uma saída válida para o *master* na Figura 12.

Figura 13 – Resultado do teste para instância executada no *branch* refatorações.



Fonte: Elaborada pelo autor.

A Figura 13 mostra o erro da execução do código-fonte para o *branch* que foi modificado e erros foram inseridos. Nesse caso, o erro identificado foi de falha na execução do sistema, porém a ferramenta é capaz também de dizer se instâncias que foram dadas como entrada sem falha de execução do sistema quebraram algum requisito funcional do *software*. Um exemplo de saída com erro é apresentado na Figura 14.

Figura 14 – Exemplo de saída com quebra de restrição.

VEÍCULOS

Quantidade de veículos usados: 2

Quantidade de veículos ociosos: 0

ID Grupo	Placa	Peso limite	Peso ocupado	Tempo limite de serviço	Tempo serviço ocupado	Inicio Almoço	Fim Almoço	Atraso almoço	Antecipação almoço
27, 2, 31	245	499999,5	0	5	191,764	12	13	0,5	0,5
24, 22, 5,	196	499999,5	0	5	192,526	12	13	0,5	0,5

CLIENTES ATENDIDOS

Quantidade: 561

ID Grupo	ID	X	Y	Demanda de peso	Tempo de serviço	Núm. visitas limite	Kilometragem atua	Expediente de início	ní-cio atendimento	Expediente de fim	Fim atendimento
0	1	1,0958246	,247240	0	10	0	27,367	8	8	18	17,913
0	5139167	51,092326	,25067	0	30	0	0,762	8	8,03	18	8,53
24	41013695	0,8246825	,88519	0	30	0	65,98	8	9,1	18	9,6
24	34508425	0,8256425	,88418	0	120	0	66,196	8	9,61	18	12,61
24	46733115	0,8197225	,87485	0	120	0	67,914	8	12,65	18	14,65
24	48802280	,7899665	,725145	0	0	0	91,66	8	15,05	18	15,05
24	54276860	,7928825	,725838	0	15	0	92,126	8	15,06	18	15,31
24	28728390	,7959935	,733757	0	21	0	93,45	8	15,35	18	15,7
24	17207835	0,8664425	,87289	0	30	0	117,712	8	16,1	18	16,6
25	1	1,0958246	,247240	0	10	0	16,242	8	8	18	17,721
25	4897355	51,091302	6,23023	0	0	0	2,738	8	8,07	18	8,07
25	4748715	-51,09062	6,23155	0	0	0	2,971	8	8,08	18	8,08
25	3835554	51,090232	6,23172	0	21	0	3,034	8	8,08	18	8,43
25	5171571	51,085912	6,23197	0	30	0	3,706	8	8,45	18	8,95
25	2892935	51,085942	6,23165	0	15	0	3,755	8	8,95	18	9,2
25	2983827	51,086242	6,23147	0	15	0	3,81	8	9,2	18	9,45
25	3822591	51,086982	6,23077	0	21	0	3,968	8	9,46	18	9,81
25	4249494	51,087142	6,23025	0	15	0	4,053	8	9,81	18	10,06
25	673043	51,087892	6,22943	0	120	0	4,225	8	10,07	18	12,07
25	3934698	-51,088	6,22942	0	15	0	4,242	8	12,07	18	12,32

Fonte: Elaborada pelo autor.

A Figura 14 mostra um exemplo onde as restrições de almoço não foram respeitadas, logo, o sistema está dando uma saída inválida e que foi apontado pelo teste de caixa preta como um erro. Ao observar a Figura 14, é possível ver que os valores das células na planilha que estão marcadas com vermelho não respeitam nem o horário de inicio e fim para o almoço e também o tempo máximo permitido para atraso e antecipação dele.

Por fim, a ferramenta foi utilizada durante as refatorações, mas também pode ser utilizada para validar qualquer mudança feita no código-fonte pelos desenvolvedores, onde essas alterações possam resultar em um comportamento diferente do esperado.

4.3 Testes unitários

O desenvolvimento manual de alguns testes de unidade ajudaram ainda na criação de outros. Segundo Rojas *et al.* (2016), testes unitários pré-existentes podem auxiliar na geração de mais testes. Ainda mais quando os testes previamente existentes foram escritos manualmente, porque eles geralmente envolvem cenários de testes mais complexos e refletem usos e interações comuns de objetos.

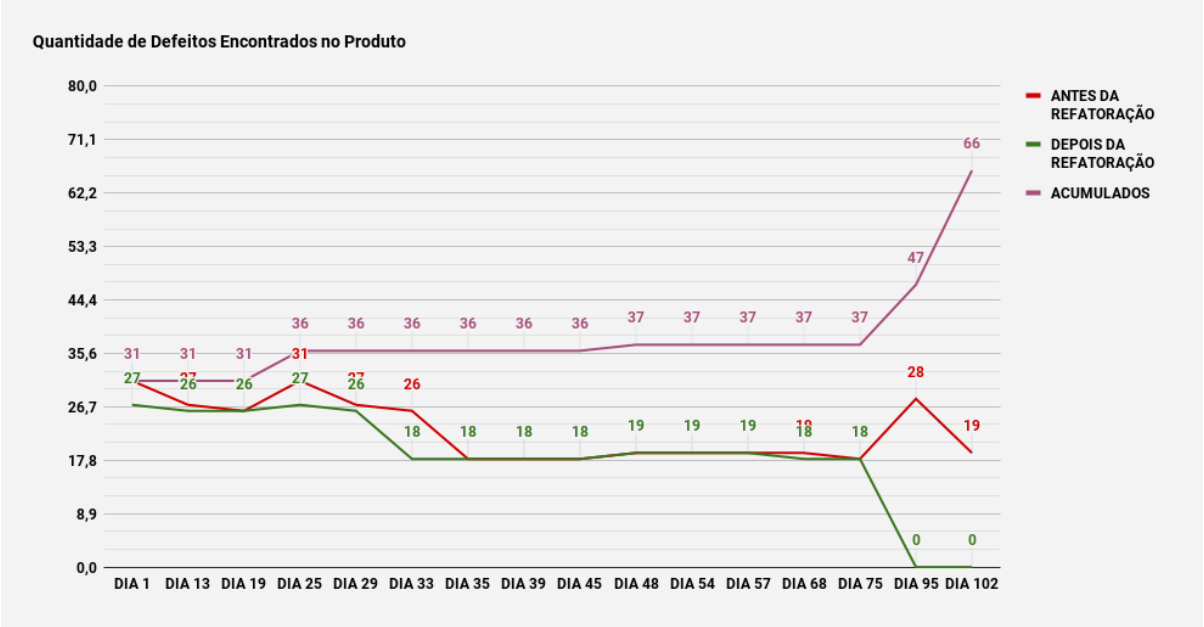
Para analisar os dados quantitativos coletados a partir dos erros identificados pelos testes unitários, foram utilizadas três métricas e analisados os gráficos observando o declínio de

alguns indicadores e o aumento de outros que estão relacionados com as métricas definidas para os testes unitários na Seção 4.1.

4.3.1 Quantidade de defeitos encontrados no produto

Essa métrica foi de grande importância para avaliar a quantidade total de erros que estavam sendo identificados pelos testes unitários em cada período e também para avaliar a redução do número de erros e a correção deles. A Figura 15 mostra o gráfico da QDP, onde é possível verificar a redução da quantidade de erros identificados à medida que iam sendo corrigidos.

Figura 15 – Quantidade de defeitos encontrados no produto pelos teste unitários.



Fonte: Elaborada pelo autor.

Através do gráfico, é possível ver como a quantidade de erros identificadas declina conforme as correções dos erros identificados eram feitas. É possível ver também que em algumas datas a quantidade de erros aumenta, quando comparada com o estado do indicador na data anterior. Isso é decorrente de erros inseridos pelas refatorações e também pelos *merges* feitos com o repositório principal do sistema. Com isso, foi realizando também a medição de quantos erros seriam acumulados caso as correções não estivessem sendo feitas, teríamos um acumulo de 66 erros que violam as restrições das unidades do sistema.

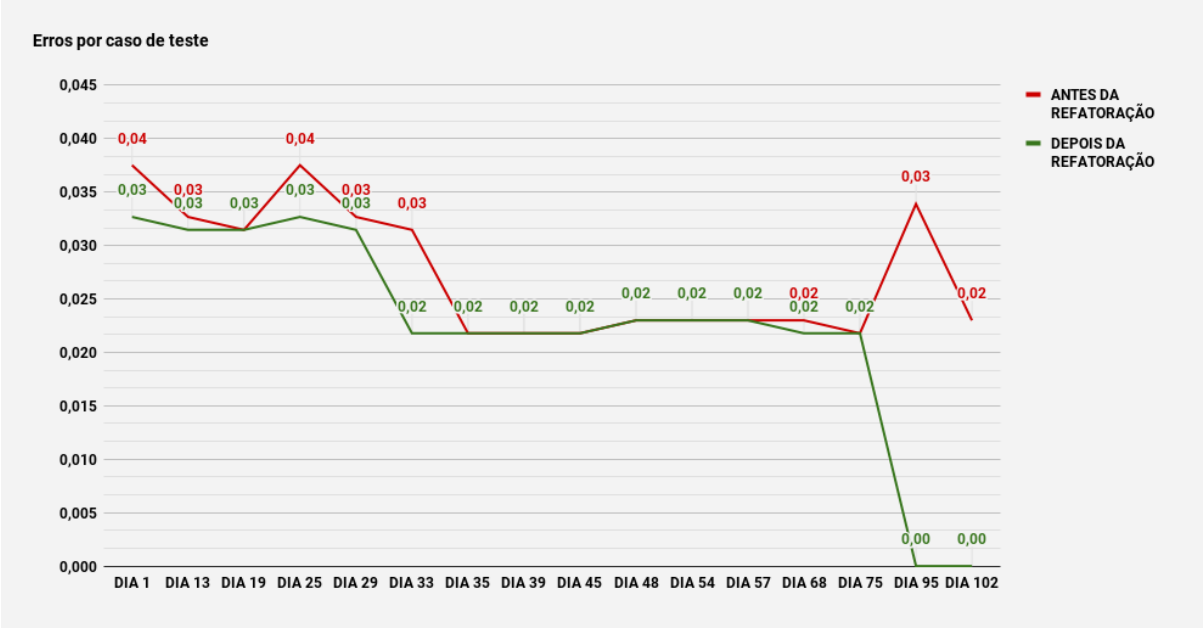
Analisando o gráfico e comparando o inicio e fim das medições é possível notar a diferença na quantidade de erros identificados, onde no inicio eram 31 erros e na última medição

foi possível reduzir a 0, o valor ideal para garantir que tudo está funcionando como deveria nas classes do sistema.

4.3.2 Erros por caso de teste

Essa métrica foi utilizada por se tratar de um sistema grande e complexo, onde dificilmente seria possível alcançar uma cobertura de testes unitários muito alta sem que grande esforços fossem concentrados para esse propósito. Com isso, ela tem a finalidade de relacionar a quantidade de erros identificados pelos testes unitários em cada período com a quantidade de casos de testes que o sistema possuía também nessa determinada data. A Figura 16 mostra o gráfico da métrica ECT, onde é possível verificar a redução da quantidade de erros identificados à medida que iam sendo corrigidos.

Figura 16 – Quantidade de erros por caso de teste.



Fonte: Elaborada pelo autor.

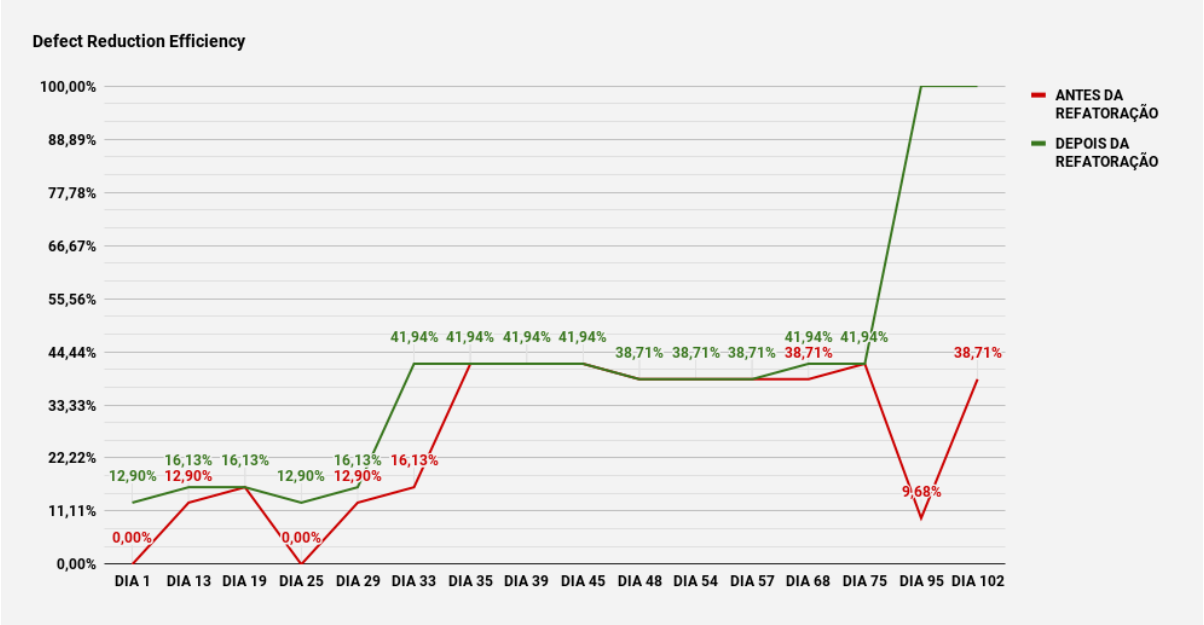
Por meio do gráfico apresentado na Figura 16, é possível ver que para cada caso de teste em média aproximadamente 0,025 erros identificados pelos testes unitários estavam presentes no sistema durante todo o período de refatorações. Porém, no último período todos os testes unitários estavam validados. Contudo, isso não quer dizer que nenhuma unidade do sistema está livre de erros, pois isso depende também da cobertura dos casos de teste e por isso essa métrica é importante para avaliar a quantidade de erros em ralação a quantidade de casos de teste. Assim, é possível ainda perceber que ao aumentar a quantidade de casos de testes

impactaria em uma maior cobertura e a também na possibilidade de que mais erros tivessem sido identificados.

4.3.3 Eficiência de redução de defeitos

Essa métrica foi utilizada também para os testes unitário com a finalidade de avaliar a eficiência na redução dos erros identificados. A Figura 17 mostra o aumento da DRDE no gráfico ao passo que as correções dos defeitos eram feitas.

Figura 17 – Eficiência de redução de defeitos identificados pelos teste unitários.



Fonte: Elaborada pelo autor.

Por meio de uma observação do gráfico apresentado na Figura 17, é possível ver que logo no inicio das refatorações já existia uma quantidade significativa de erros identificados pelos testes unitários, isso é decorrente da implementação de casos de testes que ao serem desenvolvidos já encontravam falhas nas implementações das unidades que não estavam sendo tratadas. Assim, o desenvolvimento dos casos de testes foi também uma forma de melhorar a estrutura das classes do sistema adaptando elas para que possíveis erros não acontecessem, um exemplo disso são os tratamentos necessários em métodos acessadores de uma classe Java.

Analisando o gráfico e comparando o inicio e fim das correções, no último período quando todos os defeitos identificados pelos testes unitários foram corrigidos a métrica DRDE alcançou o valor de 100%, atingindo o valor esperado quando se tratando de erros identificados pelos testes de unidade, pois esses defeitos diferente dos identificados pelas ferramentas de

análises estáticas são mais críticos e não dependem da avaliação do desenvolvedor para saber que tal erro é um problema que pode até mesmo impactar em funcionalidades do sistema.

5 CONCLUSÕES E TRABALHOS FUTUROS

Considerando as leituras realizadas para elaboração deste trabalho, pode-se tomar conhecimento que a preocupação com a qualidade de *software* tem se tornado cada vez maior em função do aumento na utilização de *softwares* e das exigências dos usuários que buscam antes de tudo sistemas de confiança, eficazes, eficientes e de boa qualidade.

De acordo com Terra e Bigonha (2008), o processo de V&V possui duas abordagens complementares para a verificação e análise do sistema que são as inspeções e testes de *software*.

Ainda segundo Terra e Bigonha (2008):

> O teste de *software*, que é uma técnica dinâmica, é a principal e mais utilizada técnica de V&V. Contudo, a inspeção de *software* vem sendo largamente utilizada pelo simples fato de as pesquisas demonstrarem que os defeitos encontrados por ela são completamente diferentes daqueles encontrados pelo teste de *software*. Logo, é recomendada a utilização destas técnicas em conjunto no processo de V&V.

Este trabalho apresentou diferentes técnicas de V&V que podem ser utilizadas de forma complementar para se obter um *software* de qualidade. Como também, foram propostas métricas de teste de *software* que podem ser utilizadas para controle de processo de qualidade das correções de erros, e também foi proposto um processo para utilização das técnicas que pode ser utilizado tanto para o desenvolvimento quanto para manutenção de *software*.

A escolha de quais técnicas podem ser aplicadas a cada organização ou *software* pode variar de acordo com restrições de custo, prazo e dos recursos disponíveis para o desenvolvimento de um sistema. Contudo, mesmo identificando e removendo tantas falhas quanto possível, encontrar todas as falhas é quase impossível. Os testes não podem durar para sempre e devem seguir alguma estratégia para alcançar a qualidade, aumentar o ciclo de vida do *software* e atender as restrições de recursos e prazos estipulados de forma eficiente.

O processo proposto funcionou bem e pode ser utilizado durante a manutenções, refatorações e desenvolvimento do *software* como um processo básico de V&V. Vale ressaltar, que para alcançar objetivos maiores em relação a qualidade de *software* é preciso a utilização de um processo mais robusto, uma equipe de testes maior e técnicas mais específicas para cada sistema. Contudo, ainda assim, para uma empresa de desenvolvimento de software que não utiliza nenhum processo e/ou técnica de V&V o processo proposto mostrou-se eficiente na melhoria principalmente de um dos principais atributos de qualidade, que é a manutenibilidade do *software*.

Durante o estudo foi utilizada como técnica de inspeção de *software* a análise estática de código com apoio de quatro ferramentas, já para verificação dinâmica foram utilizadas técnicas como o teste de caixa-preta e o teste unitário. As ferramentas de análise estática mostraram-se eficientes na remoção de defeitos para melhoria do entendimento do código-fonte, atendendo bem às expectativas para esse propósito. Além das inspeções automatizadas o teste de caixa-preta também desempenhou um papel importante no processo e mostrou ser eficiente para validação de mudanças no *software*, ela identificou alterações errôneas no código que impactaram nas funcionalidades do sistema alterando o comportamento esperado da saídas do *software*. Assim, foi possível realizar as mudanças com maior segurança de que as funcionalidades teriam o comportamento esperado. Contudo, das três técnicas utilizadas, o teste unitário foi a que teve menos eficiência na identificação de erros, acusando poucas vezes mal funcionamento das unidades dos sistema. Porém, não é possível afirmar que os testes de unidade não têm relevância no processo, uma vez que os erros identificados por eles dependem muito de que tipo de erro foi introduzido no código e também levar em consideração se todas as áreas em que os defeitos foram inseridos estão totalmente cobertas pelos testes unitários. Ainda assim, as técnicas utilizadas quando analisadas em conjunto, o que é o objetivo principal deste trabalho apresentaram ser eficientes na melhoria da qualidade do sistema, é possível afirmar ainda que elas podem se complementar muito bem e ainda abre espaço para questionamentos com sua utilização em conjunto com outras técnicas que não estavam no escopo deste estudo.

As dificuldades encontradas para realização deste trabalho foi principalmente o desenvolvimento dos testes unitários, por ser um sistema complexo e relativamente grande para o tamanho da equipe de testes. Como também, a avaliação da precisão das técnicas de teste de caixa-preta e também do teste unitário, onde, controlar o processo de avaliação dessas técnicas seriam mais complicado devido à dificuldade de inserção da ferramenta e do processo no ambiente de desenvolvimento do *software*, pois, o estudo foi realizado em um *branch* diferente do que estava sendo utilizado pelos desenvolvedores, como também, teriam um grande impacto na produção da equipe.

Pode-se dizer que este trabalho é de extrema importância, pois contribui para o conhecimento das atividades de teste de *software* e dá uma ideia de como as técnicas de V&V podem ser utilizadas em conjunto por organizações de desenvolvimento de *software*. Assim, os resultados da pesquisa foram satisfatórios, pois foram alcançados os objetivos propostos.

Como trabalhos futuros sugerem-se a utilização de mais técnicas de V&V de forma

complementar; aplicar um processo mais robusto e completo com controle através de métricas para avaliar todas as técnicas; desenvolvimento de testes unitários com maior cobertura do sistema; estudar aplicação de mais ferramentas de testes automatizados; utilizar documentações de teste como parte do processo.

REFERÊNCIAS

AMARAL, D. do; GOMES, R.; JESUS, R. P. de; ARAUJO, T. M. de; GOULART, E. E. Metodologias de teste de software. **Revista de Informática Aplicada**, Santo André, v. 5, n. 2, p. 38–49, 2010.

ARAÚJO FILHO, J. E. de; COUTO, C. F. de M.; SOUZA, S. J. de; VALENTE, M. T. Um estudo sobre a correlaçao entre defeitos de campo e warnings reportados por uma ferramenta de análise estática. In: **IX Simpósio Brasileiro de Qualidade de Software**. Belém: Sociedade Brasileira de Computação – SBC, 2010. p. 9–23.

AURUM, A.; PETERSSON, H.; WOHLIN, C. State-of-the-art: software inspections after 25 years. **Software Testing, Verification and Reliability**, Wiley Online Library, v. 12, n. 3, p. 133–154, 2002.

BOEHM, B. W. Software engineering. **IEEE Transactions on Computers**, IEEE COMPUTER SOC, Los Aalamitos, v. 25, n. 12, p. 1226–1241, 1976.

BOEHM, B. W. Software engineering: R&D trends and defense needs. **Research directions in software technology**, MIT Press, Cambridge, v. 1, 1979.

BOWEN, J. B. A survey of standards and proposed metrics for software quality testing. **Computer**, v. 12, n. 8, p. 37–42, Aug 1979. ISSN 0018-9162.

COLLINS, E.; DIAS NETO, A.; LUCENA JUNIOR, V. F. de. Strategies for agile software testing automation: An industrial experience. In: **Computer Software and Applications Conference Workshops**. Izmir: IEEE, 2012. p. 440–445.

COSTA JÚNIOR, M.; ANDRADE, S.; DELAMARO, M. Automatização de teste de software com ênfase em teste de unidade. In: AIRES, K. R. T.; MOURA, R. S. (Ed.). **II Escola Regional de Informática do Piauí**. Teresina: Sociedade Brasileira de Computação – SBC, 2016. cap. 6, p. 121–143.

DELAMARO, M.; JINO, M.; MALDONADO, J. **Introdução ao teste de software**. 2. ed. São Paulo: Elsevier Brasil, 2017.

FRASER, G.; ARCURI, A. Evosuite at the SBST 2017 tool competition. In: **10th International Workshop on Search-Based Software Testing (SBST'17) at ICSE'17**. Buenos Aires: IEEE, 2017. p. 39–42.

GAROUSI, V.; ZHI, J. A survey of software testing practices in canada. **Journal of Systems and Software**, ELSEVIER SCIENCE INC, New York, v. 86, n. 5, p. 1354–1376, May 2013.

HECHT, H.; STURM, W.; TRATTNER, S. Reliability measurement during software development. In: **Computers in Aerospace Conference**. Los Angeles: American Institute of Aeronautics and Astronautics, 1977. p. 1453.

KARHU, K.; REPO, T.; TAIPALE, O.; SMOLANDER, K. Empirical observations on software testing automation. In: **International Conference on Software Testing Verification and Validation**. Denver: IEEE, 2009. p. 201–209.

KERZAZI, N.; KHOMH, F. Factors impacting rapid releases: an industrial case study. In: **Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement**. New York, NY, USA: ACM Press, 2014. (ESEM '14, 61), p. 1–8.

LARSEN, K. G.; MIKUCIONIS, M.; NIELSEN, B.; SKOU, A. Testing real-time embedded software using uppaal-tron: An industrial case study. In: **Proceedings of the 5th ACM International Conference on Embedded Software**. New York, NY, USA: ACM, 2005. (EMSOFT '05), p. 299–306.

LAZIC, L.; MASTORAKIS, N. Cost effective software test metrics. **W. Trans. on Comp.**, World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, v. 7, n. 6, p. 599–619, jun. 2008.

LOURIDAS, P. Static code analysis. **IEEE Software**, IEEE, v. 23, n. 4, p. 58–61, July 2006.

MEDEIROS, J. E. R. de. **Estudo Comparativo de Ferramentas de Análise Estática de Código**. Dissertação (Graduação em Engenharia de Software) — Departamento de Informática e Matemática Aplicada, Universidade Federal do Rio Grande do Norte, Natal, nov 2017.

MISRA, S.; FERNáNDEZ, L.; COLOMO-PALACIOS, R. A simplified model for software inspection. **Journal of Software: Evolution and Process**, John Wiley & Sons, New York, v. 26, n. 12, p. 1297–1315, 2014.

PAULA FILHO, W. de P. **Engenharia de software**. 3. ed. Rio de Janeiro: LCT, 2009.

PEZZÈ, M.; YOUNG, M. **Teste e análise de software: processos, princípios e técnicas**. 1. ed. Porto Alegre: Bookman Editora, 2008.

PRESSMAN, R. S. **Engenharia de Software**. 7. ed. Porto Alegre: McGraw Hill Brasil, 2009.

ROJAS, J. M.; FRASER, G.; ARCURI, A. Seeding strategies in search-based unit test generation. **Software Testing, Verification and Reliability**, v. 26, n. 5, p. 366–401, 2016.

SILVA MELO, A. C. da; FERREIRA FILHO, V. J. M. Sistemas de Roteirização e Programação de Veículos. **Pesquisa Operacional**, SOBRAPO, v. 21, n. 2, p. 223–232, jul 2001.

SOMMERVILLE, I. **Engenharia de software**. 8. ed. São Paulo: Addison Wesley, 2007.

SOMMERVILLE, I. **Software Engineering**. 9. ed. New York: Addison Wesley, 2010.

TASSEY, G. The economic impacts of inadequate infrastructure for software testing. **National Institute of Standards and Technology**, RTI Project, Research Triangle Park, NC, v. 7007, n. 011, 2002.

TERRA, R.; BIGONHA, R. S. Ferramentas para análise estática de códigos java. In: **III Encontro Brasileiro de Teste de Software**. Recife: EBTS, 2008. p. 1–5.

VONKEN, F.; ZAIDMAN, A. Refactoring with unit testing: A match made in heaven? In: **2012 19th Working Conference on Reverse Engineering**. Kingston: IEEE, 2012. p. 29–38.

WIKLUND, K.; ELDH, S.; SUNDMARK, D.; LUNDQVIST, K. Impediments for software test automation: A systematic literature review. **Software Testing, Verification and Reliability**, v. 27, n. 8, 2017.

WILKERSON, J. W.; NUNAMAKER, J. F.; MERCER, R. Comparing the defect reduction benefits of code inspection and test-driven development. **IEEE Transactions on Software Engineering**, v. 38, n. 3, p. 547–560, May 2012.

YIN, R. K. **Estudo de Caso: Planejamento e Métodos**. 2. ed. São Paulo: Bookman editora, 2001.